I am the author of the thesis entitled

Building Reliable Distributed Systems

submitted for the degree of

Doctor of Science

This thesis may be made available for consultation, loan and limited copying in accordance with the Copyright Act 1968.

Full Name.......... Wanlel Zhou..................................................
(Please Print)

Signed ............... ..............................................................

Signature Redacted by Library

Date.................. 2/7/2002................................................
Consultation of Thesis

Please sign this form to indicate that you have used this thesis in accordance with the *Access to Thesis* form signed by the author of this thesis.

<table>
<thead>
<tr>
<th>NAME (please print)</th>
<th>SIGNATURE</th>
<th>DATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Andrew Baker</td>
<td>Signature Redacted by Library</td>
<td>10/9/03</td>
</tr>
</tbody>
</table>

[Signature Redacted by Library]
BUILDING RELIABLE DISTRIBUTED SYSTEMS

By

Wanlei Zhou
Associate Professor in Computing
School of Computing and Mathematics, Deakin University

B.Eng. (Computer Science and Engineering)
(Harbin Institute of Technology, China, 1982)

M. Eng. (Computer Science and Engineering)
(Harbin Institute of Technology, China, 1984)

Ph.D. (Computer Science)
(The Australian National University, Australia, 1991)

A document submitted in partial fulfilment of the requirements for the degree of

Doctor of Science

Deakin University

September 20, 2001
Declaration

I certify that the thesis entitled "Building Reliable Distributed Systems" submitted for the degree of Doctor of Science is the result of my own research, except where otherwise acknowledged.

I also certify that authors' contributions to all jointly authored papers listed in this thesis are equally distributed among all authors. That is, my contribution to a single authored paper is 100%, to a paper of two authors is 50%, to a paper of three authors is 33.33%, and so on.

Wanlei Zhou, PhD.,
Associate Professor of Computing
Deputy Head, School of Computing and Mathematics
Deakin University, Melbourne, Australia
Acknowledgements

I would like to thank Professor Andrzej Goscinski for his encouragement and help in preparing this thesis. As the Head of School of Computing and Mathematics, Deakin University, Professor Goscinski has been very supportive in my research, and my research collaboration with him has been very productive and fruitful. I would also like to thank Professor Richard Russell, Dean of Faculty of Science and Technology, Deakin University for his encouragement during my preparation of this thesis.

I have studied and worked in a number of institutions and companies and would like to express my appreciations for their support in my research activities. These institutions and companies include: Deakin University (Melbourne, Australia), National University of Singapore (Singapore), Monash University (Melbourne, Australia), Apollo/Hewlett-Packard Research Labs (Cholmsford, MA, USA), HighTech Computers P/L (Sydney, Australia), The University of Electronic Science and Technology of China (Chengdu, China), and Harbin Institute of Technology (Harbin, China). I would also like to thank some of my postgraduate students, in particular Changqui Chen, Mingjun Lan, and Shui Yu, for helping me photocopying my papers from a large number of books, journals and proceedings.

Finally, I would like to thank my wife Ling Wang and my two sons, Lingdi and Andi, for their understanding and support.

Wanlei Zhou, PhD.,
Associate Professor of Computing
Deputy Head, School of Computing and Mathematics
Deakin University, Melbourne, Australia
Contents

Declaration i
Acknowledgements ii
Contents iii
1. Executive Summary 1
1.1. Research Achievements 1
1.2. Building Theoretical Models for Fault-Tolerant Computing 2
1.3. Developing Tools for Building Reliable Distributed Systems 3
1.4. Constructing Reliable Distributed Systems 4
2. List of Complete Publications 5
3. Publications Submitted with This Thesis 12


Processing, pp. 538-545, IEEE Press, Los Alamitos, CA, USA, held in Singapore, June 1996.


[249]


[263]


[269]


[287]


[301]


[309]


[315]


[321]


[333]


[Goscinski, Hobbs, and Zhou 97] Andrzej Goscinski; Michael Hobbs, and Wanlei Zhou (Editors), Proceedings of the IEEE third International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP’97), (The Proceedings was edited as a book. We divided all papers into 12 Chapters and wrote a 13- pages Introduction on parallel processing, an Introduction for each chapter and an editorial comment for each paper), World Scientific, Singapore, 1997, 765 pages.


[Hepner and Zhou 97] Philip Hepner and Wanlei Zhou, "Integrating Heterogeneous Databases: A Distributed Model", Proceedings of the IEEE third International Conference on Algorithms and


1. Executive Summary

My application for the Doctor of Science (DSc) degree is based on my outstanding track record in research. This document is organized in three parts: In this part (Part 1) I summarize my research achievements. Part 2 presents a complete list of my refereed publications. My major research achievements are presented in Part 3 as a collection of selected publications, representing a substantial contribution to the field of reliable distributed systems.

1.1. Research Achievements

My outstanding research achievements are demonstrated by a large number of publications in first-class international refereed journals and first-class international refereed conference proceedings. Through them I have built new knowledge and added to existing knowledge for the research areas I work in.

I began publishing in English from 1989. Since then I have published more than 80 papers, with many of them featuring in first-class international refereed journals (such as The Computer Journal, The Journal of Systems and Software, Computer Communications, IEE Proceedings – Communications, and ACM Applied Computing Review) and first-class international refereed conference proceedings. These first-class international conferences include IEEE International Conference on Parallel and Distributed Processing (PDP), International Conference on Technology of Object-Oriented Languages and Systems (TOOLS), IEEE International Conference on Algorithms and Architectures for Parallel Processing (ICA3PP), International Conference on Parallel and Distributed Systems (PDCS), and IEEE/ACM International Symposium on Cluster Computing and the Grid (CCGrid). Since 1996, I have lifted my number of publications to more than 8 papers a year. My publications and research outcomes have influenced other researchers, led to further results, helped to solve industry and business problems, and shaped the directions of research areas of Reliable Distributed Computing.

I have been carrying out theoretical, experimental and practical research in the area of Distributed Systems. In particular, I have been concentrating my efforts in Reliable Distributed Computing, aiming to provide a steady, reliable source of computing power and data repository. To achieve this ambitious aim, I have created new models and extended existing models for fault-tolerant computing. These models have improved the ways we understand and design reliable computing systems; I have developed new tools for building reliable distributed systems to overcome the complexity of developing reliable distributed applications; and constructed a number of reliable distributed systems as “proof of concepts” or as real applications. These systems have either become a showcase for my research achievements or an invaluable part of real production systems. My research contributions have formed new knowledge as part of reliable distributed computing and have enriched existing knowledge. I have been recognized internationally as an outstanding researcher in the area of distributed systems. In particular, my major contributions in research include:

- Building theoretical models for fault-tolerant computing.
• Developing tools for building reliable distributed systems.
• Constructing reliable distributed systems.

1.2. Building Theoretical Models for Fault-Tolerant Computing

Building a fault-tolerant system is a difficult and challenging task, mainly due to the lack of effective models to describe the complexity of fault-tolerant requirements and to guide the development process. My major contributions to tackle this challenge are the creation of a number of effective fault-tolerant models to deal with various requirements and simplify the development process. These models include a twin-server model, the RPC (remote procedure call) transaction model, the mobile transaction model, the reactive system model, an object-oriented design pattern, and a number of fault-tolerant algorithms and protocols. These models have improved the ways we understand and design reliable computing systems. The following papers show my contributions in theoretical research of fault-tolerant models:

<table>
<thead>
<tr>
<th>Paper</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Zhou 90e]</td>
<td>31</td>
</tr>
<tr>
<td>[Zhou 92]</td>
<td>37</td>
</tr>
<tr>
<td>[Zhou 94c]</td>
<td>95</td>
</tr>
<tr>
<td>[Zhou 95b]</td>
<td>117</td>
</tr>
<tr>
<td>[Zhou 95e]</td>
<td>123</td>
</tr>
<tr>
<td>[Zhou 96c]</td>
<td>147</td>
</tr>
<tr>
<td>[Zhou 96d]</td>
<td>151</td>
</tr>
<tr>
<td>[Zhou 97c]</td>
<td>193</td>
</tr>
<tr>
<td>[Zhou and Eide 98]</td>
<td>231</td>
</tr>
<tr>
<td>[Zhou and Gosciniski 97b]</td>
<td>249</td>
</tr>
<tr>
<td>[Zhou and Gosciniski 98]</td>
<td>263</td>
</tr>
<tr>
<td>[Zhou and Gosciniski 99a]</td>
<td>269</td>
</tr>
<tr>
<td>[Zhou and Holmes 99a]</td>
<td>309</td>
</tr>
<tr>
<td>[Zhou and Holmes 99b]</td>
<td>315</td>
</tr>
<tr>
<td>[Zhou and Jia 99b]</td>
<td>333</td>
</tr>
<tr>
<td>[Zhou and Molinari 89]</td>
<td>339</td>
</tr>
<tr>
<td>[Zhou and Molinari 90a]</td>
<td>349</td>
</tr>
<tr>
<td>[Zhou and Molinari 91a]</td>
<td>359</td>
</tr>
<tr>
<td>[Zhou and Molinari 96]</td>
<td>271</td>
</tr>
<tr>
<td>[Zhou and Wang 96]</td>
<td>389</td>
</tr>
<tr>
<td>[Zhou and Wang 00]</td>
<td>395</td>
</tr>
<tr>
<td>[Zhou and Wang 01c]</td>
<td>407</td>
</tr>
<tr>
<td>[Zhou and Zhong 98]</td>
<td>427</td>
</tr>
<tr>
<td>[Chen and Zhou 00a]</td>
<td>431</td>
</tr>
<tr>
<td>[Chen and Zhou 00c]</td>
<td>437</td>
</tr>
<tr>
<td>[Chen, Zhou, and Yu 01]</td>
<td>445</td>
</tr>
<tr>
<td>[Gosciniski, Hobbs, and Zhou 97]</td>
<td>453</td>
</tr>
<tr>
<td>[Gosciniski, Ip, Jia, and Zhou 00]</td>
<td>469</td>
</tr>
<tr>
<td>[Gosciniski and Zhou 99]</td>
<td>473</td>
</tr>
<tr>
<td>[Hepner and Zhou 97]</td>
<td>495</td>
</tr>
</tbody>
</table>
1.3. Developing Tools for Building Reliable Distributed Systems

Another difficulty in developing reliable computing systems is the lack of easy to use and efficient tools for supporting the development process. To build a reliable distributed system, the developer must not only deal with the complex problems of distributed systems when all the components are well, but also the more complex problems when some of the components fail. The challenge here is to develop tools that enable developers to concentrate on their own application domain by freeing them from the difficulties of dealing with fault-tolerant issues. My major contributions in this area include the development of a number of tools for building reliable distributed systems, such as a fault-tolerant RPC system, the replica group interface definition language (RGIDL), a Web-based database architecture, a debugger/analyser for distributed programs, etc. By using these tools developing reliable applications is no longer a painful process. The following papers show my contributions in developing tools for building reliable distributed systems:

<table>
<thead>
<tr>
<th>Paper</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Zhou 90d</td>
<td>25</td>
</tr>
<tr>
<td>Zhou 93a</td>
<td>43</td>
</tr>
<tr>
<td>Zhou 93b</td>
<td>57</td>
</tr>
<tr>
<td>Zhou 94a</td>
<td>61</td>
</tr>
<tr>
<td>Zhou 94b</td>
<td>89</td>
</tr>
<tr>
<td>Zhou 95a</td>
<td>105</td>
</tr>
<tr>
<td>Zhou 96a</td>
<td>131</td>
</tr>
<tr>
<td>Zhou 96b</td>
<td>139</td>
</tr>
<tr>
<td>Zhou 96e</td>
<td>159</td>
</tr>
<tr>
<td>Zhou 97b</td>
<td>185</td>
</tr>
<tr>
<td>Zhou 99a</td>
<td>203</td>
</tr>
<tr>
<td>Zhou and Gosinski 97b</td>
<td>249</td>
</tr>
<tr>
<td>Zhou and Gosinski 99a</td>
<td>269</td>
</tr>
<tr>
<td>Zhou and Molinari 96</td>
<td>271</td>
</tr>
<tr>
<td>Zhou and Wang 01b</td>
<td>401</td>
</tr>
<tr>
<td>Wang and Zhou 99a</td>
<td>537</td>
</tr>
<tr>
<td>Wang and Zhou 99b</td>
<td>541</td>
</tr>
<tr>
<td>Wang, Zhou, and Jia 01</td>
<td>547</td>
</tr>
<tr>
<td>Yu, Chen, and Zhou 01</td>
<td>583</td>
</tr>
</tbody>
</table>

1.4. Constructing Reliable Distributed Systems

I have constructed a number of reliable distributed systems to serve either as a "proof of concepts" or as real production systems. In experimental computer science, models, theories and tools are of no value if no prototypes or real
systems are constructed based on these models, theories and tools. Some of the important reliable distributed systems I have constructed include a reliable server for the RHODOS distributed operating system, a fault-tolerant heterogeneous database system, a reliable Web-based database system, a fault-tolerant mobile transaction management system, etc. These systems have either become a showcase for my research achievements, or an invaluable part of real production systems. The following papers show my contributions in constructing reliable distributed systems:

<table>
<thead>
<tr>
<th>Paper</th>
<th>Page No.</th>
</tr>
</thead>
<tbody>
<tr>
<td>[Zhou 90b]</td>
<td>13</td>
</tr>
<tr>
<td>[Zhou 90c]</td>
<td>19</td>
</tr>
<tr>
<td>[Zhou 94a]</td>
<td>61</td>
</tr>
<tr>
<td>[Zhou 96e]</td>
<td>159</td>
</tr>
<tr>
<td>[Zhou 97a]</td>
<td>171</td>
</tr>
<tr>
<td>[Zhou 97c]</td>
<td>193</td>
</tr>
<tr>
<td>[Zhou 99a]</td>
<td>203</td>
</tr>
<tr>
<td>[Zhou 99b]</td>
<td>215</td>
</tr>
<tr>
<td>[Zhou, Augar, and Casey 01]</td>
<td>223</td>
</tr>
<tr>
<td>[Zhou and Eide 98]</td>
<td>231</td>
</tr>
<tr>
<td>[Zhou and Goscinski 97a]</td>
<td>243</td>
</tr>
<tr>
<td>[Zhou and Goscinski 97b]</td>
<td>249</td>
</tr>
<tr>
<td>[Zhou and Goscinski 99b]</td>
<td>287</td>
</tr>
<tr>
<td>[Zhou, Hepner, and Wang 96]</td>
<td>301</td>
</tr>
<tr>
<td>[Zhou and Jia 99a]</td>
<td>321</td>
</tr>
<tr>
<td>[Zhou and Zhang 00]</td>
<td>415</td>
</tr>
<tr>
<td>[Chen and Zhou 00d]</td>
<td>441</td>
</tr>
<tr>
<td>[Wang, Zhou, and Jia 01]</td>
<td>547</td>
</tr>
<tr>
<td>[Wu, Tian, and Zhou 00]</td>
<td>559</td>
</tr>
<tr>
<td>[Wu, Tian, and Zhou 01]</td>
<td>571</td>
</tr>
<tr>
<td>[Zhong and Zhou 98a]</td>
<td>589</td>
</tr>
<tr>
<td>[Zhong and Zhou 98b]</td>
<td>593</td>
</tr>
</tbody>
</table>
2. List of Complete Publications

Since 1989, I have published more than 80 papers in international refereed journals and international refereed conference proceedings. From 1984 to 1988, I also published 7 papers in Chinese computing journals and one book in Chinese. Below I list all my refereed publications, ordered by the year of publication.

2001


2000


1999


1998


1997


1996


1995


1994


1993


1992


1991


1990


1999

1984-1988 (Published In Chinese)
3. Publications Submitted with This Thesis

In this part I present my major research contributions as a collection of 67 papers selected from 96 refereed publications. These publications represent my substantial contribution to the field of reliable distributed systems.

Authors’ contributions to all jointly authored papers listed in this thesis are equally distributed among all authors. That is, my contribution to a single authored paper is 100%, to a paper of two authors is 50%, to a paper of three authors is 33.33%, and so on.
Prototyping and Debugging Remote Procedure Call Programs

W. Zhou
Department of Computer Science
Australian National University, Canberra ACT

In this paper, the design and implementation of a software tool for prototyping and debugging remote procedure call programs is described. The tool has two parts: a prototyping generator which can greatly reduce the burden of programmers, and a distributed debugger which can record and replay the events of the RPC program's execution. We believe that tool will be very helpful for NCS-based distributed programming.

1. INTRODUCTION

A distributed program consists of many processes, even programs, running on a loosely-coupled computer network. The essential part of such a program is the communication among its co-operating parts. Two kinds of communication models exist: the message passing based model and the remote procedure call (RPC) based model [3]. The RPC based communication model allows a programmer to call a procedure located at a remote computer in the same manner in which a local procedure is called. From now on when we mention distributed programming, we mean the programming using RPC model as the communication method unless we specify explicitly.

Prototyping and debugging distributed programs are extremely difficult. The main reasons are the concurrency and communications among the concurrent parts of a distributed program and the nondeterministic behaviors caused by these. Although completely automatic generation of general distributed programs is not feasible today, automatic generation of prototypes for some special area is possible. We choose the RPC model because of its special advantages [6, 2].

One of the major techniques used for improving debugging process is debugging with trace of program execution [5]. In this approach, no reproduce behavior is needed. The generated trace is no longer nondeterministic and can be analyzed in any controlled ways that a programmer prefers. But the generating of the trace may be very costly in time and space, and also the events in the trace are still not fully ordered. Some techniques are needed to analyze the program trace. Sometimes a tool doing this is called a monitor. We choose this method to debug the programs produced by the prototyping generator.

The research base we used is the Network Computing System (NCS) [1, 7] (NCS and Network Computing System are trademarks of Apollo Computer Inc.). In this system, a distributed program functionally has two pieces: the application piece and the RPC piece. Again, the RPC piece can be divided into two parts: server part and client part. Each part can be located on any host in the network. Usually, a server part manages an object, and a client part accesses the object by using the remote procedure calls provided by the server. An RPC-based distributed program (in short, an RPC program) may consist of several server and client parts, and all these parts work together concurrently on a single task. A program part (PP) can fork to several processes if necessary. Figure 1 is the abstract structure of an RPC program. It is the RPC piece of an RPC program which makes it totally different with a traditional program.

Although NCS provides some facilities (such as NIDL [1] compiler) to reduce the difficulty for distributed programming, the programming process for NCS is still much more complex compared with traditional programming. When all programs of the whole RPC system are connected, the programmer is also responsible to make sure that they are correct. This may be very difficult if the whole system has been set up. Because we have little assistance for testing the correctness of the RPC part.

In this paper, the design and implementation of a software tool for prototyping and debugging RPC programs is described. The tool has two parts: a prototyping generator which can greatly reduce the
burden of programmers, and a distributed debugger which can record and replay the events of the RPC program’s execution. We believe that tool will be very helpful for NCS-based programming.

2. ARCHITECTURE OF THE PROTOTYPING GENERATOR

Figure 2 is the architecture of our prototyping generator. Through the user interface, a programmer provides to the system several server definition files (one for each server) and their RPC procedures. Then the generator produces appropriate prototyping programs for both server and client sides. These programs can be executed on the execution system (which includes the debugger) for testing, or can be taken as proper RPC program prototypes if the tests success. After the testing execution, the debugger can replay the execution of the program in user-controlled manner.

The generator includes the following parts:

- Server Program Generator (SPG). For each server, a programmer writes a server definition file. According to the file, the SPG generates a server stub and a server program. The server program will include interfaces to RPC procedures, server stub, RPC-library, and client program. The generated server programs are ready to run.

- Sequential Client Program Generator (SCG). For each server, the SCG generates a client stub and a client drive program. The client drive program includes interfaces to client stub and the appropriate server program. If the programmer choose debug during the program generating, the client drive program will also include testing interface, and can then be used to test each of the remote procedure provided by the server sequentially. Otherwise, the drive program has to be linked with a user-provided application program for execution.

- Parallel Client Program Generator (PCG). For each server, the PCG generates a drive program which can make use of any number of remote procedures provided by the server concurrently. Depends on the selection, the parallel drive program may or may not include testing interface.

- Distributed Client Program Generator (DCG). For a group of servers, the DCG generates a drive program which can make use of any number of remote procedures provided by these servers. This drive program will include interfaces to all these servers. Also it may or may not include testing interface depending on selection.

Apparently, DCG includes all functions of PCG, and PCG includes all functions of SCG. Usually, we first use SPG to generate a server program, and then use SCG to generate a testing program (by selecting debug) for the server. In that case, we can test the server procedures one by one to make sure that they are correctly working. After that, the PCG is used to generate a parallel testing program for the server. By using this program, the concurrent execution of the server RPC procedures can be tested. If all servers are generated and tested, both sequentially and concurrently, we can use the DCG to generate a distributed testing program for all these servers, and observe the execution of all these servers when running on different hosts. After all these tests passed, the drive programs can be re-generated by de-selecting debug selection and can be used to link with real application interfaces. At this time, only command version of
the user interface is supported. We are planning to add a window-based user interface into the system.

Use of the generator can greatly reduce the burden of programming. For example, a server definition file for a server with two remote procedures (each has two simple type parameters) is about 25 lines long. According to this file, the generated programs and their approximate sizes (lines) are:

<table>
<thead>
<tr>
<th>Program</th>
<th>Size</th>
<th>NIDL</th>
</tr>
</thead>
<tbody>
<tr>
<td>NIDL file</td>
<td>30</td>
<td>NO</td>
</tr>
<tr>
<td>Server program</td>
<td>140</td>
<td>NO</td>
</tr>
<tr>
<td>Server stub</td>
<td>300</td>
<td>YES</td>
</tr>
<tr>
<td>Stub include file</td>
<td>40</td>
<td>YES</td>
</tr>
<tr>
<td>Global include file</td>
<td>40</td>
<td>NO</td>
</tr>
<tr>
<td>Client program</td>
<td>310</td>
<td>NO</td>
</tr>
<tr>
<td>Client stub</td>
<td>280</td>
<td>YES</td>
</tr>
<tr>
<td>Remote procedure frames</td>
<td>40</td>
<td>NO</td>
</tr>
<tr>
<td>Make file</td>
<td>20</td>
<td>NO</td>
</tr>
</tbody>
</table>

The NIDL field tells that if the program (file) is generated by the NCS/NIDL compiler. By filling in the remote procedure frames (that is, defining what the remote procedures will do) and using the make utility, the generated program can be executed.

3. PREPARATIONS FOR THE DEBUGGER DESIGN

When a distributed system is established, we want to know if it does the right job. In general, we can trust the generated programs. But the remote procedures were programmed by programmers and may include some errors. Even if these procedures are tested one by one, we want to know their behaviours when executed concurrently or from remote hosts. So, a debugger is needed. Before describing the architecture of the debugger, we present several definitions which are important in our discussion.

Definition 1. The name of an event (event name) is an "unique ID" (UID), defined as a string of 42 characters and an optional user attached affix. The format is

```

```

where tttttttttttttt is the timestamp; rr is a random number; h1 ... h7 are the 7-byte host ID; and nnnnnn is the sequential number of the event within its process. All of them are in hexadecimal digits. The affix is an optional name (a character string) which is defined by users.

If en is an event name, we use en.t to denote the timestamp, en.r to denote the random number, en.h to denote the host, en.n to denote the sequential number, and en.a to denote the affix of event name en, respectively. Any event can be uniquely defined, identified, and accessed by using its event name.

Definition 2. A preliminary event e is defined as a pair (f, m). Where f is called a fact and m a message. A fact is a thing which happens during a program's execution. That is, an occurrence of an action of the program. Please notice that not all facts of preliminary events are interesting to a programmer, but they may happen during the program's execution. A fact can be, for example, the creation and destroy of a process, the issuing and return of an RPC call, or an enquiring of a server's location, and so on. A message is the information attached to the fact. For example, the parameter values of an RPC call, the return information of an RPC return, the answer of a server location enquiring or even the process status.

Definition 3. Let E = {e} be the set of all events of an RPC program. If event e1 causes the occurrence of event e2, or e2 immediately follows the occurrence of e1 within the same process, we say that e1 is the predecessor of e2, and e2 the successor of e1. Especially, if e1 and e2 are in different processes, we call e1 a remote predecessor of e2 and e2 a remote successor of e1. This is denoted as e1 ≤ e2.

If we think there is a global time over all events, then e1 ≤ e2 means that e1 happens before e2. In that case, it is naturally that if e1 ≤ e2 and e2 ≤ e3, then e1 ≤ e3, and if e1 ≤ e2 and e2 ≤ e1 then e1 = e2.

We further define that e1 ≤ e2. Then the pair (E, ≤) is a partially ordered set. If a, b : E and a ≤ b holds,
then we may write this as \( b \geq a \). Suppose that \( a \neq b \) and \( a \leq b \); then we write \( a < b \).

**Definition 4.** Let \( e_1, e_2 : E \) and \( e_1 = (f_1, m_1), e_2 = (f_2, m_2) \). By \( f_1 \ast f_2 \) we mean that the happening of \( e_2 \) follows the happening of \( e_1 \), that is, \( e_1 \leq e_2 \) holds. By \( f_1 + f_2 \) we mean that we cannot tell which of \( e_1 \) and \( e_2 \) happened first, that is, there is no predecessor relation exists between these two events. We can view \( \ast \) as sequential and \( + \) as concurrent. By \( f_1 \cap f_2 \) we mean that both \( e_1 \) and \( e_2 \) occur. By \( f_1 \cup f_2 \) we mean that either or both \( e_1 \) and \( e_2 \) occur. So we have

\[
    f_1 \cap f_2 = \begin{cases} 
    f_1 \ast f_2 & \text{if } e_1 \leq e_2 \\
    f_2 \ast f_1 & \text{if } e_2 \leq e_1 \\
    f_1 + f_2 & \text{otherwise}
    \end{cases}, \quad f_1 \cup f_2 = f_1 \cap f_2 \text{ or } f_1 \text{ or } f_2.
\]

Similarly, by \( m_1 \ast m_2 \) we mean that message \( m_1 \) is followed by \( m_2 \) and by \( m_1 + m_2 \) we mean that two messages are independent each other.

**Definition 5.** If \( e_1 = (f_1, m_1) \) and \( e_2 = (f_2, m_2) \) are events, then

\[
e_1 \cap e_2 = (f_1 \cap f_2, m_a), \quad e_1 \cup e_2 = (f_1 \cup f_2, m_b),
\]

\[
e_1 \ast e_2 = (f_1 \ast f_2, m_1 \ast m_2), \quad \text{and } e_1 + e_2 = (f_1 + f_2, m_1 + m_2)
\]

are also events. Where

\[
m_a = \begin{cases} 
    m_1 \ast m_2 & \text{if } e_1 \leq e_2 \\
    m_2 \ast m_1 & \text{if } e_1 \leq e_2 \\
    m_1 + m_2 & \text{otherwise}
    \end{cases}, \quad m_b = \begin{cases} 
    m_a & \text{if } e_1 \cap e_2 \\
    m_1 & \text{if } e_1 \text{ and not } e_2 \\
    m_2 & \text{if } e_2 \text{ and not } e_1
    \end{cases}.
\]

We call these new events as **combined events**. Combined events are usually defined by programmers, and are interested to users. Some properties can be derived from the above definitions [8].

4. **ARCHITECTURE OF THE DEBUGGER**

4.1. **Architecture**

![Architecture of Distributed Monitor](image)

The debugger consists of a controller and a group of managing servers. The controller has two main parts: an user interface (including commands interpreter and I/O) and a filter. While a managing server is consisted of a server (MS) and an event database. Each host which has monitored program parts on it has a managing server. The controller can be invoked at any host. By communicating with each related MS, the controller can present the monitored results to the user. Of course, several controllers can be invoked by several users simultaneously. Figure 3 illustrates the architecture.
Three steps are used during debugging. At the monitoring step, all events that happened on one host are monitored by the local MS and recorded into the local event database. All preliminary events including RPC calls and executions as well as process forks and exits in both client and server program parts are monitored, and an user can also define through the Event Definition File (EDF) some combined events and let the debugger record them. After all events are recorded, the programmer uses the ordering step to order events. At that time, each MS exchanges remote predecessor / successor information through the controller and has all remote relationship ordered. Then, local predecessor / successor relationship is established by each MS over its local event database respectively. The last step is replaying. By combining the results on all related event databases, the filter can present the execution trace of the distributed program to the user. The display speed and view contents are controlled by the user through the command interpreter. Two kinds of user interfaces are provided. One is a command level interface, another is a graphics interface. An user can use commands to order the debugger to start a remote / local program part and let the debugger to monitor the program part. Or the user can have the debugger to monitor a program part which has been in executing.

5. TRACING ANALYSIS

5.1. Ordering Events

The following steps are used to establish the partial ordering among all events:

1. Remote procedure call related predecessors. When a remote procedure call related event happens (for example, the issuing or ending of an RPC call), it will cause the happening of an event which belongs to another process (and also possibly, on another host). In that case, the first event is changed (by the debugging library) to carry not only the original information, but also the event’s name. On the other hand, the second event is also changed (by the debugging library) to not only receive the original information, but also the first event’s name, and this name is stored by the local MS as the predecessor of the second event. All remote procedure call related predecessor events can be stored in that way.

2. Process fork related predecessors. When a process fork event happens, it will cause a new process be setup and executed. This event is changed (by the debugging library) to carry the name of the event, and the first event of the new process will use the carried name as its predecessor event.

3. Combined event related predecessors. When an event with an affix definition happens, it will cause the controller to evaluate the related combined event expressions. So, the name of the event is sent to the controller and stored as one of the predecessors of the related combined events.

4. Form all remote successors. In (1), (2), and (3), all remote predecessors will be established after the termination of the monitored programs. The remote successors are built by each involved MS and the controller at this moment. Each MS checks all events in its local database. If its event e has a remote predecessor named f, then the MS will responsible to store e as f’s successor. It is easy if e and f are in the same database (for example, the fork events). Otherwise f, h is used to locate the MS it belongs to and f’s successor will be stored by the communication of these two MS’s through the filter.

5. Form all other successors and predecessors. All the events within a process are ordered by their sequential numbers, and their predecessors / successors are stored by using this order. In one process, the predecessor of event e is event d if d.n is immediately less than e.n. And the successor of e is event f if f.n is immediately greater than e.n. This ordering is performed by each MS concurrently.

5.2. Process Replay

The replaying of an RPC program is to replay its program parts, sequentially or concurrently, on the controller’s terminal. For each program part, we can use the partial order relation to build an event graph and use this graph for replaying and analysis. The building process is simple. If a, b : E and a < b, then we have an arc from a to b.
The only method that two program parts (PP, server or client program) can communicate each other is by using RPCs. If we do not care any RPCs at this moment, then the event graph of each PP of an RPC program is a weakly connected acyclic graph. Now we consider the RPC communications among these program parts. In that case, the event graphs of all program parts of the RPC program will be connected by the predecessor relation of those RPC calls. So, usually the event graph of the RPC program is also a weakly connected acyclic graph.

If we add the successor relation into the event set and build the graph, the result event graph will be a bi-directed graph. In that case, it is easy to control the event accesses during replay. That is, if the graph is weakly connected (this is the usual case), we can access any event from the beginning of the program, or even from any event. In our debugger, an user can control the event replay speed (or even single step), view any program part of an RPC program, view the details (such as messages and fact fields) of an event, and dump the results into files. The events can be displayed in both predecessor or successor orders. By using the graphics interface, one can display several event graphs on the controller terminal in several windows, and view the replays of these event graphs concurrently. We will not mention the details.

6. FURTHER RESEARCH DIRECTIONS AND ACKNOWLEDGEMENT

In this paper we presented a prototyping generator and a distributed monitor for RPC programs. By using that tool, the programming burden will be greatly relieved.

It seems that many researches can be carried out in this area. Firstly, the current version can only produce C language prototypes, while NCS can support multiple languages. So a multi-language interface is valuable. Secondly, the reusable modules database [4] is one of the powerful techniques in prototyping. If we can build such a database and find some methods to access and use the modules, the prototyping system will be more powerful. Thirdly, a source language level debugging is needed in the execution system for debugging the prototypes. Fourthly, the expansion of the prototyping system to RPC procedures and application part of an RPC program is also valuable. We are currently investigating some of these issues.

The author would like to thank Dr. Brian Molinari and Dr. John Smith for general discussion and commenting on the research project.

References

Computer-Aided Prototyping of Distributed Programs: A Case Study

W. Zhou
Department of Computer Science
Australian National University, Canberra ACT

In this paper, we present a case study that illustrates a method of prototyping one kind of distributed programs — the remote procedure call based distributed programs. The distributed system is a distributed calendar which consists of many programs and several databases across a local area network. After the introduction and background description, the prototyping system is presented with the illustration of examples from the distributed calendar system. Then, the prototyping process is described. The last section concludes the paper.

1. INTRODUCTION

As we know, for the past twenty years or so, software system development has been based on the software life-cycle model (or, waterfall model)\(^2\). This model essentially advocates that software projects should consist of a number of distinct phases\(^6\). These are: specification, design, implementation, testing, operation and maintenance. This model has been modified by a lot of researchers since its inception. But the central idea remains unchanged, that is, all variations keep the linear structure, and each phase begins only when the previous phase has been completed.

The life-cycle model works very well when an application is both well understood and supported by previous experience, such as embedded software and real time control systems\(^7\), but for many other applications, such as distributed information system (IS) applications, it is inappropriate and has many deficiencies which are too serious to be ignored. The reason is that the naive assumptions made by life-cycle model is no longer true for distributed IS application development.

In a distributed IS application, the program consists of several co-operative parts that are distributed over a computer network and closely cooperate to achieve a common goal. Examples of this are distributed decision making systems, distributed database management systems, and distributed text handling systems. In such systems, the communications among the cooperative parts are very important and complex. There are many uncertainties within a distributed system. Even the goals that a user wants to achieve by using the distributed system techniques are usually ambiguous and conflicting because the complexity of the system (I'll know what I want when I see it)\(^8\). This situation is quite common during IS development, and adds more uncertainties to the development of a distributed IS application.

Rapid prototyping has been suggested to overcome the deficiencies of life-cycle model for developing IS applications\(^9\). When the interactive is an important part of the developing system, when the requirements cannot be completely specified at the beginning, or when there are too many uncertainties about the proposed system, the rapid prototyping is a suitable model for development.

Although we can not generate a complete valuable program automatically from very high level specifications, the prototype generation in some application fields is feasible. Program prototyping has been investigated by a lot of researchers. But to the best of our knowledge, there are little prototyping systems deal with distributed programs. In this paper, we present a case study that illustrates a method of prototyping one kind of distributed programs — the remote procedure call (RPC) based distributed programs.

2. BACKGROUND

2.1. Network Computing System

The basis we used for our distributed programming is Apollo's Network Computing System\(^1,9\) (NCS).\(^7\) NCS is a successful commercial RPC-based distributed computing tool. In this system, a distributed program functionally consists of two kinds of programs: server program and client program. Each program

\(^{\dagger}\) NCS and Network Computing System are trademarks of Apollo Computer Inc.
can be located on any computer in the network. Usually, a server program manages an object, such as a database, and a client program accesses the object by using the remote procedures provided by the server.

2.2. The Distributed Calendar System

The work was to create a distributed calendar system for a computer company. The company has an LAN composed of hundreds of Apollo workstations and we assume all of them have NCS installed. At the beginning, it was intended to manage the following functions:

(1) Given the data of a meeting, we want that meeting be arranged in the respective person’s calendar if all participants of the meeting are free during the meeting period;

(2) Allow a person to set some personal reminders into his own calendar.

At first we identified the following concepts:

- Caller. A person that issues the meeting.
- Callee. One of the persons that a caller wants to have a meeting with.
- Meeting data. A message which includes the caller's name, the callee's name, meeting time interval, and a brief description of the meeting purpose (see the calendar.def file at the next section).
- Calendar database. If a user has a meeting, there will be an entry in the database which contains the meeting data and can be accessed by the user's login name and other keys.
- The processing of reminders are the same as of calendars except that the meanings of the fields are changed to the user's name, the reminder's time, and the reminder message.

3. PROTOTYPING SYSTEM

3.1. Prototyping System

At first, let us look at the general structure of a distributed application. By "distributed", we mean that the programs of the application is located on several computers within a local area network (LAN). A distributed application consists of several local programs and several remote programs. Usually, a local program interfaces with the user, manages the local application process, and performs the communication between the local program and other related remote programs. A remote program usually manages an object (e.g., one part of a distributed database), performs the operations required by other programs, and manages the communications. So, we can divide a distributed application into three parts: The user interface deals with the interactions between the local program and the user; The distributed frame performs the communications among all the co-operative parts over the LAN; And the application modules manage the objects and performing operations.

Based on the observation, Our development model can be described as in figure 1. Different tools are provided for each prototyping activity. For example, when prototyping the user interface, an interface generator is used to generate the interface program from an easy-learning definition file. When prototyping the distributed frame, a knowledge-based prototyping tool is provided to generate the frame. For application modules, only database management operations are provided at this moment. All these prototypes are relatively independent and can be tested separately with the participating of users. This gives the users a good chance to clear the ambiguity in their mind for the system requirements and to rethink and reorder their needs. After each part of the prototype is tested, their combination forms a working prototype of the whole program and can be tested again by users and developers. Further changes can be made easily because the computer-aided generators are available.

3.2. User Interface Generator

The user interface generator has two functions. It can generate the window-oriented or command-oriented interface screen to accept the user commands. It can also generate the data entry screen according to the data format and layout definitions. The definition files are self-explanatory. For example, if the user defined the system menu as:

```
menu window
```
Then, the generator will produce a menu program which will display at first the menu as fig.2(a) if linked with the related functions (for menu testing, all the related functions, such as create() in the menu definition can be just dummy functions). If the user selected MEETING, the menu screen will change to fig.2(b). When Create is selected, the associated function create() will be executed. If there is any changes, the developer can simply change the definition file and (probably) the related dummy functions, and re-generate the source files. This only costs minutes. The generation of data enter screen is similar.

3.3. Distributed Frame Generator

A knowledge-based distributed frame prototype generator has been built to generate the distributed frame based on NCS\(^{11}\). It uses a sequence of queries as well as RPC function template files to obtain information of the distributed frame from the developer. Fussy decision strategy is employed for selecting implementation modules of the frame. The generator will produce all the remote and local communication modules as well as the testing programs for those co-operative modules. If compiled and linked with NCS library, the distributed frame will be ready for execution. Of course, the RPC functions of each server will only return a information specifying that it works normal, because only function templates were defined in the testing program. Knowledge and transaction mechanism of the generator are implemented by using CLIPS\(^{2}\), and C is used to help the implementation. We will not specify the details here. The following is the query process for server calendar (see the next section for its purpose):

What is the program convenience consideration:
(best/good/moderate/low/do-not-care) best
What is the program speed consideration:
(highest/high/moderate/low/do-not-care) high
What is the program memory consideration:
(smallest/small/moderate/large/do-not-care) moderate
What is the program complexity consideration:
(smallest/small/moderate/large/do-not-care) moderate
How many different servers? 1
What is the No. 1 server name? calendar
Where does the server locate:
  (global/local/well-known/do-not-care) global
Which protocol is to be used:
  (ip/dds/do-not-care) ip
Is the user provide server UUID (yes/no)? no
How many multi-accessing to the server allowed:
  (1-5/do-not-care) 5
Is remote shutdown of server allowed (yes/no)? yes
Is server exception handling included (yes/no)? yes
Usage of RPC in client:
  (sequential/parallel/do-not-care) parallel
Is client exception handling included (yes/no)? yes
Name of the remote procedure specification file: calendar.def

The remote procedure specification file calendar.def may be defined as:

BEGIN
  constant int MAXNAMELEN 20;
  constant int MAXVALLEN 13;
  constant int MAXMTLEN 255;
  constant int MAXRESLEN 20;
  typedef struct {
    char username[MAXNAMELEN];
    char start[MAXVALLEN];
    char end[MAXVALLEN];
    char caller[MAXMTLEN];
    char comment[MAXMTLEN];
  } DISCAL_REC;

Procedures:
  Name: find_a_cal;
    Param: [in] char name[MAXNAMELEN];
    Param: [in] char start[MAXVALLEN];
    Param: [out] DISCAL_REC rec;
  Name: find_all_cal;
    Param: [in] char name[MAXNAMELEN];
    Param: [out] DISCAL_REC recs[MAXRESLEN];
    Param: [out] int ret_num;
  Name: add_a_cal;
    Param: [in] DISCAL_REC rec;
  Name: del_a_cal;
    Param: [in] char name[MAXNAMELEN];
    Param: [in] char start[MAXVALLEN];
  Name: change_a_cal;
    Param: [in] char name[MAXNAMELEN];
    Param: [in] char start[MAXVALLEN];
    Param: [in] DISCAL_REC rec;
END

The generator will produce several programs according to these input. These programs can be tested independently. After the testing, they can form the remote and local communication modules of the system.
3.4. Application Module Generator

Currently we only implemented a database application generator which can produce simple database-oriented programs. The main purpose here is to generate RPC functions of servers. The algorithm implementation has to rely upon the developers at this moment.

For example, if the following is the input to the application generator, then the appropriate program of "add an entry to database calendar" function will be generated.

Name: add_a_cal is an ADD_ENTRY program for database calendar;

Param: DISCAL_REC rec;

These programs can be tested separately. Then can be easily connected to the distributed frame programs.

4. PROTOTYPING PROCESS

According to the initial goals described in section 2.2, the first prototype is developed and tested. It consists of a registry database and a calendar database. Each database is managed by a server (remote program). Local programs (application programs) interface with users and access databases by using the remote procedures provided by servers. The above generators are used to generate these programs.

After the initial testing, the user identifies that there should be a meeting room database because the room conflicting is also a problem. So that a room database is added, and a new field meeting_room is added into the calendar.def file. That causes the distributed frame prototype be re-generated to have the meeting room server working co-operatively with the calendar server. Several new modules are added into the user interface prototype to fit the new database; And some new application modules are added. Also, the first requirement is changed to:

(1). Given the data of a meeting, we want that meeting be stored in the meeting room database and be arranged in each related person's calendar if the meeting room and all participants of the meeting are free during the meeting period;

Next, the user discovers that after a meeting is set up, there is no way to cancel it. But in the real life there might be such cases. So, a function is added to the application prototype that a caller can cancel a meeting he/she has entered. That is, the third requirement is added:

(3). Allow a caller to delete a meeting which he/she entered from meeting room database and all participants' calendars.

This causes the menu description file be changed and programs re-generated. After further re-prototyping and testing, the user and the developer realise that there should be a function to automatically delete all old meeting data, that is, the meetings that have been held. That function then is added to the database manager. It checks the database periodically to delete all old entries. That is, the fourth requirement is added:

(4). Automatically delete all old meeting entries (that is, the meeting ending time is earlier than the current time).

When the system is put in use by some users, they find that there should be some restrictions on the issuing of meetings. That is, some priorities are needed among all users. So, the system is changed again that each user is assigned a priority number. In that case, only the users with higher or equal priorities can arrange meetings with users having lower or equal priorities. So a new field is added into the registry database to specify the user priority. At last, the system is ready for use.

5. CONCLUSIONS AND ACKNOWLEDGEMENT

Not surprisingly the computer-aided rapid prototyping method will minimise the distributed IS application development cost. For example, in a reported prototyping experiment, systems were developed at 40% less cost and 45% less effort than conventional methods. Other researchers have reported even more impressive figures. As in our example, only one man week is used for the initial prototype while it was estimated 4 man weeks before. We claim that our computer-aided rapid prototyping model is more suitable to distributed IS application than the conventional life-cycle model.
The author would like to thank David Ortmeyer and Jim Perry in Apollo Computer Inc for the discussion of the design of the distributed calendar system.

References


Fig. 1. Computer-Aided Prototyping Model

Fig. 2. The Menu Screen
On The Rapid Prototyping of Distributed Information System Applications

Wanlei Zhou
Department of Computer Science
The Australian National University
Canberra, ACT 2601, Australia

Abstract

Until recently, most developers of information system applications use the conventional life-cycle model as their guidance of software development. As the evolving of the underlying technology and expanding of the application domain, the life-cycle model is no longer a proper model for advanced information system development. After described the deficiencies of the life-cycle model for distributed IS application development, the rapid prototyping model and associated tools for developing distributed IS applications are presented in this paper. Then a case study of the new model applied to a real distributed IS application is described. From the illustrations of the example, we conclude that the new model is better than the old for distributed IS applications.

Key Words: Software Engineering, Rapid Prototyping, Information System(IS), Distributed Computing, Remote Procedure Call(RPC).

1. INTRODUCTION

1.1. The Life-Cycle Model and its Deficiencies

Computer technology has had a tremendous effect on the working lives of employees of an organization within the last two decades. Now it is generally accepted that information systems (IS) are a powerful instrument for organizational problem solving. But unfortunately, despite all the efforts of users, analysts, and developers, the development of information system remains a highly imperfect process, especially compared with the rapidly advances of the underlying computer technology and the dramatically expanding of computerized application domain.

As we know, for the past twenty years or so, software system development, as well as information system development has been based on the software life-cycle model (or, waterfall model). This model essentially advocates that software projects should consist of a number of distinct phases. These are: specification, design, implementation, testing, operation and maintenance. This model has been modified by a lot of researchers since its inception. But the central idea remains unchanged, that is, all variations keep the linear structure, and each phase begins only when the previous phase has been completed.

The life-cycle model works very well when the application is both well understood and supported by previous experience, such as embedded software and real time control systems, but for the majority of information system applications, especially for distributed information system applications, it is inappropriate and has many deficiencies which are too serious to be ignored. The reason is that the naive assumptions made by life-cycle model is no longer true for distributed IS application development.

In a distributed IS application, the information system consists of several co-operative parts that are distributed over a computer network and closely cooperate to achieve a common goal. Examples of this are distributed decision making systems, distributed database management systems, and distributed text handling systems. In such systems, the communications among the cooperative parts are very important and complex. There are many uncertainties within a distributed system. Even the goals that the user wants to achieve by using the distributed system techniques are usually ambiguous and conflicting because the complexity of the system. The system is variable, the communication is (almost) invisible, and the cooperations of the concurrent parts are unpredictable.

Because a distributed IS is available in a network, many people from many different backgrounds (instead of just experts as twenty years ago) will use it through its user interface. So the user interface of a distributed IS application is a significant component of the system and must be user friendly.

As in other IS applications, the functional requirements of a distributed IS application are usually not well understood. Even sometimes the user does not know what is really he or she wants (I'll know what I want when I see it). This situation is quite common during IS development. This adds more uncertainties to the development of a distributed IS application.

Also, there is usually a significant cultural gap between the user and the IS developer and the way they communicate: The developer is usually not an expert about the application, while the user is usually not an expert on programming and computer systems. Consequently, a user often finds it extremely hard to visualise a system by simply reading a technical system specification document, and that makes the validation of the specification a very error prone activity.

1.2. Rapid Prototyping

Rapid prototyping has been suggested to overcome the deficiencies of life-cycle model for developing IS applications. When the interactive is an important part of the developing system, when the requirements cannot be completely specified at the beginning, or when there are too many uncertainties about the proposed system, the rapid prototyping is a suitable model for development.

Several types of rapid prototyping exist. In throw-away prototyping, the prototype is to be used for a limited period, and usually is used for requirements analysis and specification. After that, it is throw away. The rapid development of the prototype is the greatest need, while the quality of the prototype is of little importance. The other is the evolutionary prototyping. Here a system grows and evolves gradually. At first, only some of the understood parts of the proposed system is developed, then the prototype evolves as the understanding of the whole system becomes clearer during the using of the first prototype. This is very suitable for gradually introducing a new system into an organization and for coping with the changes that take place within the organisation as a result of using the system. This method is the most attractive model for IS applications.
In all cases, a prototype must be a "working model" of the proposed IS application. So the challenge is to develop the related software tools that support the prototyping process. In the next section, we present several aids that help the development of distributed IS application prototypes.

Of course, rapid prototyping is not an universal desirable model. Actually we think that different software products need different developing models. But for distributed IS applications, the rapid prototyping is indeed a wise choice.

THE COMPUTER-AIDED RAPID PROTOTYPING SOLUTION

1. Structure of the Distributed IS Applications

At first, let us look at the general structure of a distributed IS application. By "distributed", we mean that the programs of the IS application are executed on several computers within a local area network (LAN). We call a user's computer as the local computer, and other computers with the IS application programs located on as remote computers. Please notice that the "local" and "remote" are relative to the user (or the program which uses the impact). Figure 1 pictures the general structure.

Fig. 1. The general structure of a distributed IS application

A distributed IS application consists of several local programs and several remote programs. Usually, a local program interfaces with the user, manages the local application process, and performs the communication between the local program and other related remote programs. A remote program usually manages an object (e.g. one part of a distributed database), performs the operations required by other programs, and manages the communications. So, we can divide a distributed IS application into three parts: the user interface deals with the interactions between the local program and the user; the distributed frame performs the communications among all the cooperative parts over the LAN. And the application modules manage the events and performing operations.

2.2. The Development Environment

From the discussion of section 1. it is clear that flexibility, not rigidity, is needed to deal with the problems encountered by distributed IS software development. Computer-aided rapid prototyping is one of the suitable ways to provide such flexibility. Based on the general structure of the distributed IS applications, the new development model can be described as in figure 2.

One of the pre-requirements of using the computer-aided prototyping model is to have all the related tools available. When user requirements changes, it is required to respond that with a new prototype ready for testing in a short time.

Different tools are provided for each prototyping activity. For example, when prototyping the user interface, an interface generator is used to generate the interface program from an easy-to-learn definition file. When prototyping the distributed frame, a knowledge-based prototyping tool is provided to generate the frame. For application modules, only database management operations are provided at this moment. All these prototypes are relatively independent and can be tested separately with the participating of users. This gives the users a good chance to clear the ambiguity in their mind for the system requirements and to rethink and reorder their needs. After each part of the prototype is tested, their combination forms a working prototype of the whole program and can be tested again by users and developers. Further changes can be made easily because the computer-aided generators are available.

Fig. 2. The computer-aided prototyping environment

2.3. User Interface Generator

The user interface generator has two functions. At first, it can generate the window-oriented or command-oriented interface screen to accept the user commands. It can also generate the data entry screen according to the data format and layout definitions. The definition files are self-explanatory. For example, if the user defined the system menu as:

```
menu window
  menu 0
  direction:  horizontal;
  position:  default;
  selections:  "MEETING",  child
  menu 1,
```
help "Enter a new meeting."
"REMEMBER", child menu 2,
help "Enter a new reminder.",
"SHOW", child menu 4,
help "Show my engagement.",
end selections
end menu 0:

menu 1
direction: vertical;
position: default;
selections:
"Create", function create(),
help "Input new meetings"
"Edit", child menu 6,
help "Modify existing meetings"
end selections
end menu 1;
/* others are omitted for saving spaces */

end menu

Then, the generator will produce a menu program which will display as first the menu as fig.3(a) if linked with the related functions (for menu testing, all the related functions, such as create() in the menu definition can be just dummy functions). If the user selected MEETING, the menu screen will change to fig.3(b). When Create is selected, the associated function create() will be executed. If there is any changes, the developer can simply change the definition file and (probably) the related dummy functions, and re-generate the source files. This only costs minutes. The generation of data entry screen is similar.

![Meeting Menu](image)

2.4. Distributed Frame Generator

The fundamental distributed system we are currently using is the Apollo's Network Computing System 1.14 (NCS) (NCS and Network Computing System are trademarks of Apollo Computer Inc.). NCS is a successful commercial RPC-based distributed computing tool. In this system, a distributed program functionally consists of two kinds of programs: server program and client program. Each program can be located on any computer in the network. Usually, a server program manages an object (as one of the remote programs in fig. 1), and a client program (as one of the local programs in fig. 1) accesses the object by using the remote procedure calls provided by the server.

A knowledge-based distributed frame prototype generator has been built to generate the distributed frame based on NCS 15. It uses a sequence of queries as well as RPC function template files to obtain information of the distributed frame from the developer. Fuzzy decision strategy is employed for selecting implementation modules of the frame. The generator will produce all the remote and local communication modules as well as the testing programs for those co-operative modules. If compiled and linked with NCS library, the distributed frame will be ready for execution. Of course, the RPC functions of each server will only return a piece of information specifying that it works normal, because only function templates were defined. Knowledge and transaction mechanism of the generator are implemented by using CLIPS 18 and C is used to help the implementation. We will not specify the details here. The following is the query process for server calendar (see the next section for its purpose):

What is the program convenience consideration:
(best/good/moderate/low/do-not-care) best
What is the program speed consideration:
(highest/high/moderate/low/do-not-care) high
What is the program memory consideration:
(smallest/small/moderate/large/do-not-care) moderate
What is the program complexity consideration:
(smallest/small/moderate/large/do-not-care) moderate

How many different servers? 1
What is the No.1 server name? calendar
Where does the server locate:
(global/local/well-known/do-not-care) global
Which protocol is to be used:
(ip/dns/do-not-care) ip
Is the user provide server UUID (yes/no)? no
How many multi-accessing to the server allowed:
(1-5/do-not-care) 5
Is remote shutdown of server allowed (yes/no)? yes
Is server exception handling included (yes/no)? yes
Usage of RPC in client:
(sequential/parallel/do-not-care) parallel
Is client exception handling included (yes/no)? yes

Name of the remote procedure specification file:
calendar.def

The remote procedure specification file calendar.def may be defined as follows:
BEGIN

class constant int NAMELEN 20;
class constant int VALUELEN 13;
class constant int MAXNAMELEN 255;
class constant int MAXARGS 20;

typedef struct {
char username[MAXNAMELEN];
char start[MAXVALUELEN];
char end[MAXVALUELEN];
char caller[MAXNAMELEN];
char comment[MAXVALUELEN];
} DISCAL_REC;

178
procedures:

Name: find_a_cal;
  Param: [in] char name[MAXNAMELEN];
  Param: [in] char start[MAXVALLEN];
  Param: [out] DISCAL_REC rec;

Name: find_all_cal;
  Param: [in] char name[MAXNAMELEN];
  Param: [out] DISCAL_REC recs[MAXRESULT];
  Param: [out] int ret_num;

Name: add_a_cal;
  Param: [in] DISCAL_REC rec;

Name: del_a_cal;
  Param: [in] char name[MAXNAMELEN];
  Param: [in] char start[MAXVALLEN];

Name: change_a_cal;
  Param: [in] char name[MAXNAMELEN];
  Param: [in] char start[MAXVALLEN];
  Param: [in] DISCAL_REC rec;

END

The generator will produce several programs according to these input.
These programs can be tested independently. After the testing, they can
form the remote and local communication modules of the system.

2.5. Application Module Generator

Currently we only implemented a database application generator
which can produce simple database-oriented programs. The main purpose
here is to generate RPC functions of servers. The algorithm implementation
has to rely upon the developers at this moment.

For example, if the following is the input to the application generator,
then the appropriate program of "add an entry to database calendar"
function will be generated.

Name: add_a_cal
  is an ADD_ENTRY program for database calendar;
  Param: DISCAL_REC rec;

These programs can be tested separately. They can be easily connected to
the distributed frame programs.

3. AN EXAMPLE

One of the the advantages of using the new development model is that
it can establish a good relationship between the user and the developer. The
user learn how to express their needs better, and the developers have an
improved understanding of the specific environment. Usually, the earlier
modules are simple and need more backtracking, while the subsequent
modules tend to be more complex and require less refinement and back-
tracking. 4

Next we describe an application of the model to solve a real dis-
tributed IS problem. The work was to create a distributed calendar system for
a computer company. The company has an LAN consisted of hundreds of
Apollo workstations and we assume all of them have NCS installed. At the
beginning, it was intended to manage the following functions:

(1). Given the data of a meeting, we want that meeting be arranged in each
related person’s calendar if all participants of the meeting are free during
the meeting period.

(2). Allow a person to set some personal reminders into his own calendar.
At first we identified the following concepts:

- Caller. A person that issues the meeting.
- Callee. One of the persons that a caller wants to have a meeting with.
- Meeting data. A message which includes the caller’s name, the
  callee’s name, meeting time interval, and a brief description of the
  meeting purpose (see the calendar.def file for the previous
  section).
- Calendar database. If a user has a meeting, there will be an entry in the
  database which contains the meeting data and can be accessed by the
  user’s login name.

- The process of reminders are the same as for calendars except that the
  meanings of the fields are changed to the user’s name, the reminder’s
  time, and the reminder message.

Based on these assumptions and requirements, the first prototype was
developed and tested. Then the user identified that there should be a meeting
room database because the room conflicts are also a problem. So that a room
database was added, and a new field meeting_room was added into the
calendar.def file. That caused the distributed frame prototype be re-
generated to have the meeting room server working co-operatively with the
calendar server. Several new modules were added into the user interface
prototype to fit the new database; And some new application modules were
added. Also, the first requirement was changed to:

(1). Given the data of a meeting, we want that meeting be stored in the
meeting room database and be arranged in each related person’s calen-
dar if the meeting room and all participants of the meeting are free during
the meeting period.

Next, the user discovered that after a meeting is set up, there is no way
to cancel it. But in the real life there might be such cases. So, a function was
added to the application prototype that a caller can cancel a meeting he/she
has entered. That is, the third requirement was added:

(3). Allow a caller to delete a meeting which he/she entered from meeting
room database and all participants’ calendars.

After further re-prototyping and testing, the user and the developer
realized that there should be a function to delete all old meeting data, that is,
the meetings that have been held. That function then was added to the data-
base manager. It checks the database periodically to delete all old entries.
That is, the fourth requirement was added:

(4). Automatically delete all old meeting entries (that is, the meeting end-
ting time is earlier than the current time).

When the system was put in use by some users, they found that there
should be some restrictions on the issuing of meetings. That is, some priori-
ties are needed among all users. So, the system was changed again that each
user was assigned a priority number. In that case, only the users with higher
or equal priorities can arrange meetings with users having lower or equal
priorities. Because there had been an user information server existing in the
LAN, an interface with that server was built and two fields were added into

179
the database managed by the user information server. These two new fields specify the user priority and the user calendar server location, respectively. At last, the system was ready for use.

4. CONCLUSIONS

Not surprisingly, the computer-aided rapid prototyping method will minimize the distributed IS application development cost. For example, in a reported prototyping experiment, systems were developed at 40% less cost and 45% less effort than conventional methods. Other researchers have reported even more impressive figures. As in our example, only one man-week is used for the initial prototype while it was estimated 4 man-weeks before. We claim that our computer-aided rapid prototyping model is more suitable as distributed IS application than the conventional life-cycle model.

One of the underlying assumptions that support the new model is the availability of different prototyping tools. Better tools are needed for further development of the new model.

References

Fault Tolerant Computing: An Improved Recursive Algorithm

Wanlei Zhou
Department of Computer Science
The Australian National University
Canberra, ACT 2601, Australia

e-mail: zhou@anu.edu.au  Phone: (062)49-3890

Abstract
In this paper, an improved recursive algorithm for fault tolerant computing is described. The original algorithm is presented by P. Agrawal. The improved algorithm uses majority voting to achieve more believable results than that of the original algorithm. Both space and time redundancy are employed dynamically in the algorithm. The algorithm is also extended to a general form. By defining different parameters, the algorithm will suit different system situations. The improved algorithm also has the original algorithm as one of its special cases. The performance of the improved algorithm is compared with the original algorithm by simulation. The results show that the improved algorithm is suitable for many circumstances.

Introduction
The advances in VLSI technology has made it cost-effective to include a large number of very powerful processors together with their local memories in a multiprocessor system. With the increased complexity of computing units (processor and its memory) as well as software used for it, the number of possible ways in which error conditions can occur in such a computing unit (CU) may increase6. The system will still work if some of its computing units failed. In this case, we have to ensure that we can still obtain correct result even if there exist faulty CUs in the system4.

The Recursive Algorithm for Fault Tolerance (RAFT)1,2 is a simple but very effective algorithm for fault tolerant computing. But the result produced by RAFT is not the most "believable" and even incorrect in some situations. As one of the main purposes of a fault tolerant system is to achieve the believable result even there are faulty units in the system3,4, more efforts are needed to improve the RAFT algorithm.

Suppose we have an CU pool with N CUs and any CU can be used in our algorithm by specifying its index number in the pool. We do not care here how these CUs are interconnected. User jobs are scheduled to CUs, and results and signatures are obtained from these CUs. A signature may be obtained by using many techniques, such as data compression from the result. The main requirement for a signature is that it should reflect the different errors in the results.

We define the believability of a result produced by a fault tolerant algorithm R with respect to a job T as

$$B^R_T = \frac{NS}{NC}$$

where NS = Number of CUs in the maximum signature match, and NC = Number of CUs used in computing job T. We usually omit T if no ambiguity will be caused. Usually, we would hope that $B^R > 0.5$. That is, majority CUs should have the same signatures.

Now suppose, for example, we want to get the correct result of job T by using RAFT system with N CUs, and we are now in the stage that $L - 1$ CUs are used by the algorithm and they all have distinct signatures. If the $L$th CU is activated and it has the same signature with one of the used $L - 1$ CUs, then the algorithm will terminate and report that result as the correct one. As L CUs have been used and only two of them have the same signatures, the believability of RAFT is

$$B^{RAFT} = \frac{2}{L}$$

(according to the algorithm, $L \geq 2$).

So, when $L$ increases, $B^{RAFT}$ decreases. Also there are $N - L$ CUs left unused which may help to produce the correct answer.
In this paper we are going to give an improved RAFT algorithm (RAFT) which uses the "majority voting" to produce more "believable" results. The RAFT can be also considered as one of the special cases of IRAFT. Let us define a voting function as follows:

\[
    r(L) = \begin{cases} 
        2 & \text{if } L = 2 \\
        \left\lceil \frac{L}{2} \right\rceil & \text{if } L \text{ is odd}
    \end{cases}
\]

where \( L \) is an integer and \( L > 2 \), and the function is undefined if \( L \) is even. If \( L \) CUs are used and the RAFT produces a correct result, then there must be \( r(L) \) CUs have the same signatures. So the believability of RAFT is \( B^{RAFT} = \frac{r(L)}{L} > 0.5 \). We also call \( r(L) \) as the exactly majority number over \( L \) if \( L > 2 \). It is evident that \( B^{RAFT} \geq B^{RAFT} \). And \( B^{RAFT} > B^{RAFT} \) if \( L \geq 5 \). Also, if there are not enough CUs left the IRAFT will produce the degraded result which includes the result of RAFT.

Description of IRAFT Algorithm

Notations and a Common Procedure

We adopt the following definitions from Agrawal:

- \( P_i \): \( i \)th CU, \( i = 1, 2, \ldots, N \)
- \( R_i \): Result of a job on the \( i \)th CU
- \( S_i \): Signature of \( R_i \) on the \( i \)th CU
- \( LF \): Set of CUs whose signatures are not match with the correct one

For the convenience of describing the algorithm, we use the following procedure to determine the maximum equal signatures from the CU set \( V \). Its implementation is easy.

procedure MaxEqual(V, t, q, m);
/* where */
V: Input parameter, Set of CUs already used in processing the job;
t: Number of CUs in V;
q: Output parameter, Number of signatures in the maximum equal signature match among CUs in V;
m: Output parameter. The smallest index of the above q CUs. */

For example, let \( V = \{ p_1, \cdots, p_3 \} \) and \( t = 5 \). If \( p_2, p_4, \) and \( p_5 \) have the same signature. Then the procedure will return \( q = 3 \) and \( m = 2 \).

Description

There are two major differences between RAFT and IRAFT. Firstly, the RAFT produces the correct result as long as two CUs produce the same signatures, while the IRAFT produces the correct result only if the majority CUs produce the same signatures. Secondly, when the signatures are not match with (or no majority match in IRAFT), the RAFT reschedules the job on to one new CU, while the IRAFT reschedules the job on to \( k \) new CUs concurrently, where \( k \) depends on the used CUs and maximum equal signatures among the used CUs. That is, the newly scheduled \( k \) CUs plus the used CUs will produce the exactly majority signature match over the number of latest used CUs if the \( k \) new CUs produce the signatures as the original maximum equal signatures.

The essential idea of IRAFT is as follows: Suppose now we have a job \( T \) for processing. At the beginning, \( T \) is scheduled to two CUs, say \( P_{i_1} \) and \( P_{i_2} \), and these two CUs work concurrently. After the results \( R_{i_1} \) and \( R_{i_2} \) and their signatures \( S_{i_1} \) and \( S_{i_2} \) generated, the two signatures are compared. If they match with each other, the algorithm terminates and the correct result is output. If they do not match, then more CUs are needed. Suppose we are now in the stage that \( r \) CUs are used and \( x \) is the maximum number of CUs which have the equal signatures. If \( x = r(i) \) (according to the algorithm, \( x \) can not be greater than \( r(i) \)), that is, the majority CUs have the same signatures, then the algorithm terminates and the correct result is output. Otherwise, \( T \) is rescheduled to \( k = t - 2 \ast x + 1 \) unused CUs at the next stage. If \( k + t > N \), the number of CUs in the system, then only the unused \( N - t \) CUs are scheduled. In this case, the degraded result is generated and the algorithm terminates.

Algorithm

Following is the description of the IRAFT algorithm.

BEGIN
Schedule job \( T \) on 2 CUs \( P_{i_1} \) and \( P_{i_2} \);
Obtain \( R_{i_1}, R_{i_2}, S_{i_1}, \) and \( S_{i_2} \);
IF \( (S_{i_1} = S_{i_2}) \) THEN /* first two match with */
Output \( R_{i_1} \) or \( R_{i_2} \) as correct result;
Exit "Job completed"
END IF;
/* more CUs are needed */
k := 1; V_1 := \{P_{i_1}, P_{j_1}\}; t := t + 1;
DO WHILE (k + t ≤ N)
BEGIN
  Schedule job T on k CUs of V_2;
  V_2 = \{P_n | n = 1, ..., k\} \AND (P_n \not\in V_1);
  Obtain R_{\alpha} and S_{\alpha}, n = 1, ..., k;
  V := V_1 \cup V_2;
  MaxEqual(V, t, q, m);
  IF q = [\frac{(k+1)}{2}] THEN /* vote succeeded */
    Output R_{\alpha} as correct result;
    Add \{P_n | (S_n \not= S_{\alpha} \AND (P_n \not\in V)\} to LF;
    Exit "Job completed";
  ELSE /* vote failed; prepare next stage */
    V_1 := V; t := t + 1;
    k := (2q + 1);
  END IF
END DO;

/* next is the case of degraded result output */
k := N - t;
Schedule job T on k CUs of V_2;
V_2 = \{P_n | n = 1, ..., k\} \AND (P_n \not\in V_1);
Obtain R_{\alpha} and S_{\alpha}, n = 1, ..., k;
V := V_1 \cup V_2;
MaxEqual(V, t, q, m);
Output (R_{\alpha}, q) as degraded result;
Add \{P_n | (S_n \not= S_{\alpha} \AND (P_n \not\in V)\} to LF;
Exit "Job completed. Degraded result."
END.

The IRAFT includes the RAFT as its special case.
If we change statement IF q = [\frac{(k+1)}{2}]
THEN ... into IF q = 2 THEN ..., and statement
k := (t - 2q + 1); into k := 1; then the resulted algorithm is the same as
RAFT. Of course the degraded result in RAFT is actually no use. As in RAFT, the collected
faulty-like CUs in set LF can be used for diagnosis.

Correctness Of The Algorithm
At first we claim that the algorithm will terminate
after a finite time of computation. There are three
termination possibilities:

Term1: When the signatures of the first two CUs match with, one of their results
is output to the user and the algorithm terminates;

Term2: When majority signatures match in the used CU set V achieved, R_{\alpha}
is output as the correct result and the algorithm terminates;

Term3: When there are not enough unused CUs left, all left CUs are used in the
last computation and the algorithm terminates after the degraded result is reported.

Because all used CUs are put into set V_1 and the newly needed CUs for job T are taken from those
unused (relative to job T) CUs (that is, each CU is only used once during a job's computation), and
there are finite N CUs in the CU pool, so if majority signatures match before all CUs are exhausted, one of the first two termination possibilities (Term1 or Term2) will be reached. If no
majority signature matches occur, the CU pool will be exhausted and the last termination possibility
will be reached.

Next we claim that the algorithm produces majority signature matches or degraded results if it terminates. We denote by V the used CU set for job T, and by S the maximum number of equal signatures of all CUs in V. It is evident that at Term1, \|V\| = 2 and S = 2. So, we have majority match. At
Term2, \|V\| = k + t and S = q = [\frac{(k+1)}{2}] = \frac{|V_1|}{2}. So, we still have majority match. At
Term3, \|V\| = N, the number of CUs in the CU pool, and the degraded result is output.

Extension Of The Algorithm
We denote the original algorithm as algorithm A, the improved algorithm as B. The performance of
the algorithm can be enhanced much more if we use three CUs at the first step. That is, three CUs
are scheduled to job T at the beginning. We denote this algorithm as algorithm C.

We can easily extend the algorithm into a more general version. Let a be the number of CUs that are used
in the first computing and comparing step, and b be the number of CUs that are used for the following voting steps. Where a is an integer and b an integer expression. We denote
the extended algorithm as A_{a,b}. Then, algorithm A can be denoted as A_{2,2}. If we change the function
r(L) to

\[
r(L) = \begin{cases} 
  L & \text{if } L = a \\
  \frac{L + 1}{2} & \text{if } L \text{ is even and } L > a \\
  \left\lfloor \frac{L}{2} \right\rfloor & \text{if } L \text{ is odd and } L > a 
\end{cases}
\]

853
and denote \( b = r(U_i) \), algorithm \( B \) can be denoted as \( A_2^2 \), and algorithm \( C \) \( A_3^1 \). The extended algorithm can be described as follows:

At first, a CUs are scheduled for job \( T \) concurrently. If all those signatures match with, one of the results is output as the correct answer. Otherwise, new CUs are needed. If we are now in the stage that \( i \) CUs are used and \( x \) is the maximum number of CUs which have the same signatures, then by solving the function \( r(i + k) = x + k \), we know that \( k \) CUs are to be scheduled in the next iteration.

Performance Simulation and Analysis

Descriptions and Assumptions

The performance of algorithm \( A \), \( B \), and \( C \) are simulated and results are compared in this section. The simulation tool is based on MacDougall's amp17. Some modifications are made in order to suite the non-queueing system characteristics of the algorithms. In order to make the simulation models more realistic, jobs entering the system are divided into two classes. Class 0 is the ordinary jobs, i.e., jobs that do not require fault tolerant computation; Class 1 is the fault tolerant jobs, and they may use algorithm \( A \), \( B \), or \( C \), respectively. We have the following descriptions and assumptions:

- Suppose there are \( N \) identical CUs in the CU pool, and they are numbered from 1 to \( N \). For any CU \( P_i \), its service discipline is FCFS. All jobs in the same class have the same service time distribution at \( P_i \), and the service time distribution is a negative exponential. We use \( \mu_i \) to denote the service rate for class \( i \) jobs \((i = 0, 1) \). We also assume that service rates are independent of the system status.

- There are two arrival streams to the system, one for each job class. We assume that all streams are Poisson streams. The arrival rate of class \( i \) jobs is \( \lambda_i \) \((i = 0, 1) \).

- Let \( T \) be a class 1 job. Let \( p \) be the probability of an CU being good with respect to \( T \), and \( E \) the set of failure modes of an CU with respect to job \( T \), \(|E| = m \). That is, if \( P_i \) fails with respect to \( T \), then the failure mode \( e_i \) can be one of the \( m \) elements in \( E \). We also assume that \( e_i \) is equally distributed in \( E \). Of course, here we will assume that the signature technique can differ all these failure modes.

Performance Metrics

If we use \( R \) to represent the algorithm used for fault-tolerance. The performance metrics we used in our simulation are:

Average Number of error results:

Denoted by \( AER^R \). This is the error results occurred during the simulation period. For algorithm \( A \), an error result will occur if two CUs produce the same failure modes; while for algorithm \( B \) and \( C \), an error result will occur if the majority signature match reached but the result is a failure mode.

Average CUs used:

Denoted by \( APU^R \). This is the average number of CUs used by algorithm \( R \) in order to obtain a result. It denotes the resource needs of algorithm \( R \).

Average response time:

Denoted by \( ART^R \), where \( i = 0, 1 \). It denotes the average response time of class \( i \) jobs when algorithm \( R \) is used in the system.

Results Analysis

The following simulation results are achieved when \( \mu_0 = 5.0, \mu_1 = 10.0, \lambda_0 = 4.0 \), and \( \lambda_1 = 12.0 \). Similar results will be obtained if different \( \mu_i \) and \( \lambda_i \) are used.

Fig. 1 is the average response time (ART) variations when the number of CUs in the pool changes. We know from the simulation that if \( N \) is large enough, the class 0 jobs will not be affected because of the different algorithms used in class 1 jobs. Also class 1 jobs will not be affected by class 0 jobs if \( N \) is large enough. This is also true from intuition. Usually a multiprocessor system is comprised of a lot of CUs, so we will not consider \( N \) any more.

One of the important performance metrics for a fault tolerant algorithm is the average number of error results (AER) occurred during the computation. It may be affected by the number of fault modes \( (m) \), or by the probability of an CU being good with respect to a job \( (p) \). Fig. 2 gives the AER comparison of the three algorithms. Fig. 2(a) is the AER variation when \( m \) changes and \( p=0.90 \) (please notice that \( m \) is expressed by loga-
We can see that AER decreases when \( m \) increases. When \( m > 1000 \), the three algorithms produce the same performance. But when \( m \) is small, algorithm B and C are much better than algorithm A. Also when \( m < 100 \) (i.e., \( g(m) < 2 \)), AER\(^C\) increases very little, while AER\(^B\) and AER\(^A\) increase sharply. So, algorithm B and C are suitable when \( m \) is median, and algorithm C is very suitable when \( m \) is small.

Fig. 2(b) is the AER variations when \( p \) changes and \( m=500 \). We can see that if \( p > 0.95 \), the three algorithms produce the same performance. But when \( p \) decreases, AER\(^A\) increases sharply. While the AER's of the other two algorithms increase very little. So, the improved algorithms (algorithm B and C) are suitable when \( p \) is low.

Fig. 3(a) is the average number of CUs used (APU) when p changes. Here we set \( m=500 \). It is seen that when \( p > 0.85 \), \( APU^A = APU^B \), and \( APU^C = APU^B + 1 \). This is because at that case, algorithms A and B will output almost all correct results at the first step of computation, and they all use two CUs at the first step. Algorithm C will also output most correct results at the first step, but it uses three CUs in that step. So we have the above relations. When \( p \) increases, APU\(^A\) increases a little, while APU\(^B\) and APU\(^C\) increase greatly. Also APU\(^B\) and APU\(^C\) keep a difference about one CU. This is true because algorithm C uses one more CU at the first step, and the two algorithms use the same majority voting strategy after the first step failed. That is, when \( p \) is small, the improved algorithms will use more resources than the original algorithm.

Fig. 3(b) is the average response time (ART) variations when \( p \) changes. Here we also set \( m=500 \). It is seen that when \( p > 0.85 \), \( ART^A = ART^B \) and \( ART^C \) is greater than \( ART^B \) a little. When \( p \) decreases, the response time of all algorithms increases and \( ART^B \) and \( ART^C \) increase faster than \( ART^A \). That is, if \( p \) is small, the improved algorithms will take much longer to obtain a result than that of the original algorithm.

Conclusions and Acknowledgement

An improved recursive algorithm for fault tolerance is described and its correctness and extension are discussed. The performance of the improved algorithm is compared with the original algorithm by simulation. The results show that when \( p \) (the probability of an CU being good with respect to a job) is small, or \( m \) (the number of fault modes of an CU with respect to a job) is small, the improved algorithm is very suitable.

When \( p > 0.85 \) and \( m > 500 \), the original algorithm is perfect.

The author would like to thank Professor G. Hellestrand for his commenting on the research.

References

A Decentralised Remote Procedure Call Transaction Manager

Wanlei Zhou

Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 0511
Email: zhouwi@isee.nus.sg

Abstract

In a centralised RPC transaction management model, when the coordinator's host fails, the transaction can not proceed. While a decentralised RPC transaction management model can cope with any host's failure. This paper describes a model for decentralised management of remote procedure call (RPC) transactions in a distributed system.

1. Introduction

In [9] and [10], we proposed a centralised RPC transaction manager. In the system, a RPC transaction is managed by one coordinator and some relevant slaves. When received a user's transaction request, the coordinator sends a start message to all the relevant slaves and waits for responses from them. If all slaves and the coordinator agree to commit the transaction then the coordinator sends a commit message to these slaves and the transaction commits. Otherwise the coordinator sends an abort message to all the slaves and the transaction aborts. Because there is a single coordinator in the commit protocol, it is called centralised.

One weakness of a centralised protocol is that when the coordinator fails, the whole transaction must be aborted. Decentralised protocols are used to solve this problem. In a decentralised protocol, all participants are considered to be coordinators and all of them execute an identical program.

The needs for transaction-oriented RPC calls are obvious. For example, in our distributed calendar application, a user usually issues a meeting which involves a group of people. Suppose that everybody in the group is a key person to the meeting, that is, if anybody cannot attend the meeting, then the meeting period must be re-arranged by the issuer. The calendar database of those participants may locate on several different hosts. So if anything goes wrong (such as the system failure of the related hosts or the time periods of some participants are already occupied and cannot be re-allocated) during the meeting arrangement call, which is a concurrent call of several RPCs, we need a method to rollback those calls that have been performed.

There have been some suggestions to maintain the atomicity of a remote procedure call. The objectives are to obtain the recoverability and the indivisibility. The recoverability requires that the overall effect of a RPC call is all-or-nothing; while the indivisibility requires that the partial effects of a RPC call are invisible to other RPC calls. Liskov and her colleagues considered the atomicity of RPC from the viewpoint of programming languages. Concepts such as guardians, actions, atomic data types, and promises are introduced to ensure the atomicity of some segments of a program. We considered the problem from the system's viewpoint by using a runtime transaction manager to manage the RPC transactions. When a RPC transaction is submitted in the system, the transaction manager will maintain its atomicity by rolling back all the failed RPCs of the transaction.

A preliminary implementation of the transaction manager has been tested on a network of DEC workstations.

All these RPC transaction management are centralised and cannot tolerate the coordinator's host failure. In this paper we present a decentralised RPC transaction model. The remainder of this paper is organised as follows. Section 2 describes the RPC transaction model. Section 3 presents the structure and algorithms of the RPC transaction management model. Section 4 describes the properties of the model, and Section 5 is the conclusions.

2. The RPC Transaction Model

Assume there are N hosts in a distributed system and they are fully connected via a communication network. These hosts are numbered from 1 to N. The communication network provides error-free, in-sequence, and guaranteed message delivery. The message delays are unknown but bounded. A host does not receive or send any message after its failure and before its recovery. The upper bound of the message delay will be used to issue a
timeout to an awaiting host. When this timeout is
detected, the awaiting host knows that the message it is
sending out does not arrive the destination, or the mes-
sage it is waiting for will not arrive, due to the failure of
the destination host. Let
\[ P = \{ p^i \mid p^i \text{ is a remote procedure call to host } i \} \text{ and } \\
D = \{ d \mid d \text{ is a data item of the object system } \} \]

Please note that two \( p^i \)'s are treated differently in \( P \) even
if they use the same program segment to make their calls.
After we make a remote procedure call, the call may
return successfully or may fail. We define the effect of a
RPC call as the processing of a data item of the object
system. Hence we can abstract the type of a RPC call as
a mapping
\[
c : P \times D \rightarrow \{OK, FL\}.
\]

**OK** This means that no error occurred during the
RPC's execution. By \( c(p^i, d) = OK \), where \( p^i \) is a RPC call and \( d \) is a data item of the system,
we mean that the RPC call was successful (i.e., the remote procedure performed its assigned
task).

**FL** This means accessing failure. By \( c(p^i, d) = FL \),
we mean that the destination server (the server that
exports the remote procedure \( p^i \)) did not perform
its assigned task. This may occur if any fatal error
happened during the RPC call's execution.

We provide a rollback operation that will reverse the call
if it happened and do nothing if it did not happen. The
operation is defined as:
\[
r : P \times D \rightarrow \{OK, FL\}.
\]

The range set \( \{OK, FL\} \) has the similar meaning as
above. We assume that a rollback operation \( r(p^i, d) \) will
roll back the effect of what \( c(p^i, d) \) has done to a particu-
lar data item \( d \) if \( c(p^i, d) \) has been performed. If \( c(p^i, d) \) has not been performed or a previous \( r(p^i, d) \)
has been successful, then we assume that further \( r(p^i, d) \)
will have no effect on the data item. Using mathematical
terminology, the mapping \( r \) is said to be an idempotent
operation. For example, if a RPC call is to "reserve the
required seat for customer X if it is free" of a flight reser-
vation system, then the rollback operation may be "clear
the reservation of customer X." If the reservation for cus-
tomer X has been cleared, the rollback operation has no
effect. So, here OK means that the RPC has been rolled
back, or it has been rolled back before by other rollback
operations, or even the RPC is not performed.

If no confusion will be caused, we may simplify a RPC
call \( c(p^i, d) \) into \( c(p^i) \), and a rollback call \( r(p^i, d) \) into
\( r(p^i) \).

We have the following assumptions:

1. All RPC operations can be rolled back. And all \( r 
\)
   mappings are idempotent operations.
2. The First-In-First-Serve (FIFS) strategy is used to
   order all operations of a server (except parallel
   operations).
3. All orphaned calls are exterminated before a host
   recovers from crash to normal operation.
4. When a RPC updates a data item, it locks the data
   item until the transaction the RPC belongs to
   finishes (commits of aborts). RPCs in other trans-
   actions that want to update the same data item will
   have FL returned.

Now we can describe our RPC transaction model. We
define a RPC transaction as
\[
T = (c(p_1, d_1), \ldots, c(p_m, d_m)),
\]
where \( c(p^j, d_j) \) \((j = 1, 2, \ldots, m)\) is a RPC call, and
\( p^i, d^i \) \( i = 1, \ldots, m \) is a data item of the
system. If there is no confusion, we also denote \( T \)
as \( T = (c(p_1), \ldots, c(p_m)) \). If \( m = 1 \), we call it a single
RPC transaction. If \( m > 1 \), we call it a parallel RPC
transaction. We also call all the hosts \( i_1, \ldots, i_m \) as relevant
hosts of \( T \).

The semantics of a RPC transaction \( T \) is that after issuing
the transaction, all \( c(p^j) \) of \( T \) will be executed, or if any
one of them fails, all executed RPCs will be rolled back.
The execution of all \( c(p^j) \) in \( T \) is in parallel if \( i > 1 \).
Some parallel primitives can be built for parallel execu-
tion of remote procedures. Sequentially executed trans-
action can be easily established from the parallel model.

3. RPC Transaction Management

3.1. The Manager

Each host in the network has a decentralised RPC transac-
tion manager (DRM) executing on it. As each host it
assigned a unique number from 1 to \( N \), the set of DRMs
in the network can be expressed as \( \{DRM_1, DRM_2, \ldots, DRM_N\} \). A DRM consists of
a managing server and three tables which are stored in the
stable storage. These tables are:

**LET Locally Executed RPC Table.** When a RPC is per-
formed by a server in the host, it is reported to the
DRM and stored into LET. The function of LET
table is to denote the executed RPCs for possibly
rollbacks. An entry of the table is defined as:

```c
typedef struct let {
    char *rpc; /* RPC call name */
    char *data; /* its data object */
} LET;

TRM Transaction and Received Messages. It contain:
the transaction message and messages received
```

38
from other relevant hosts. The first entry of the TRM contains the transaction details. Other TRM table entries are defined as:

typedef struct tm {
    int hn; /* host number of a RPC call */
    char *data; /* contain "yes" or "no" */
} TRM;

NRT Needed Rollback Table. If the DRM decides that a RPC is to be rolled back, it is put into the NRT. A NRT table entry is defined as:

typedef struct nrt {
    char *rpc; /* RPC call name */
    char *data; /* its data object */
} NRT;

3.2. Algorithms

The DRM's algorithm is described in Algorithm 1, where LET, TRM, and NRT tables and their counters are located in the stable storage. When the DRM is first invoked, all these tables are set to empty and their counters are set to 0 by the initialization function. Constant MAXENTRIES is defined as the maximum number of entries in each of these tables. The DRM forks into two "forever running" concurrent processes when it is invoked.

Algorithm 1.

LET *let_tb[MAXENTRIES];
int let_ct; /* LET table and count */
TRM *trm_tb[MAXENTRIES];
int trm_ct; /* TRM table and count */
NRT *nrt_tb[MAXENTRIES];
int nrt_ct; /* NRT table and count */
initialisation();

cobegin
    /* managing transaction */
    decen_rpc(let_tb, trm_tb, nrt_tb); /* periodically roll back entries in NRT */
    rollbck_nrt(nrt_tb);

cobend

When an incoming transaction $T$ arrives, the first function decen_rpc() stores it into the TRM table (as its first item). It then checks if the host is a relevant host of $T$. If the transaction has nothing to do with the host, it simply exits and awaits a new transaction. If the host is a relevant host, the function then performs these RPCs and stores the correspondent items into the LET table. If any of the RPCs have a non-OK return it then reports a "no" to each relevant host. The function exits and awaits a new transaction. If all local RPCs returns OK it then reports a "yes" to each relevant host. The function then waits for messages from other relevant hosts (with a time-out facility). If all messages are received and they are all "yes" then the transaction commits. The LET table is then cleaned. Otherwise the transaction aborts and all the items in LET table are sent to NRT table for rolling back. The algorithm of the function is as follows (we assume the local host is numbered $k$):

Algorithm 2.

decen_rpc(let_tb, trm_tb, nrt_tb)
    LET *let_tb[];
    TRM *trm_tb[];
    NRT *nrt_tb[];
    
    LET *a;
    TRM *b;
    1 while (TRUE) {
        i/* listen to RPC transaction calls. (suspend until a transaction call comes) */
        2 */ if more transaction calls come, they are queued up until current processing is over */
        3 listen_call(T=[c(p_1, d_1), ..., c(p_m, d_m)]);
        4 store T into trm_tb;
        5 if no $i_j = k$, $j=1, \ldots, m$
            exit;
        6 /* my host is a relevant host */
        7 for all $i_j = k$ do
            $r = c(p_j, d_j)$;
            8 if $r = FL$ /* call fails */
                report "no" to all hosts
                $i_j = k$, $j=1, \ldots, m$;
            /* abort the transaction */
                abort_t(let_tb, nrt_tb);
            exit; /* exit from the Algorithm */
        9 else /* call successes */
            a.rpc = $p_j$; a.data = $d_j$;
            LET += a; /* insert a into LET */
        10 /* now I can commit the transaction of my part */
        11 report "yes" to all hosts
            $i_j = k$, $j=1, \ldots, m$;
            if any timeout
                /* abort the transaction */
                abort_t(let_tb, nrt_tb);
            exit;
        12 receive messages from all hosts
            $i_j = k$, $j=1, \ldots, m$
            and store them into trm_tb;
        13 if all received messages are "yes"
            commit the transaction;
            cleanup let_tb;
        14 else
            /* abort the transaction */
            abort_t(let_tb, nrt_tb);
            exit;
    } /* while */
Next is the function `abort_t()` used in Algorithm 2:

```c
abort_t(let_tb, nrt_tb)
LET *let_tb[];
NRT *nrt_tb[];
{
    abort the transaction;
    store let_tb items into nrt_tb;
    cleanup let_tb;
}
```

The second function periodically rolls back all entries in the NRT table. If a rollback is successful, the entry is deleted. Next is the algorithm of the function:

Algorithm 3.

```c
rollback_nrt(nrt_tb)
NRT *nrt_tb[];
{
    while (TRUE) {
        for all b ∈ nrt_tb do
            s = r(b); /* roll back it */
            if (s == OK)
                /* if rollback succeeds, delete b */
                delete b from nrt_tb;
    }
}
```

The next algorithm is used in recovering from host failures. As we have known, all the three tables are stored in the stable memory and are safe from crashes. When recovering, the host will use these information to check if the transaction processed just before the crash is complete or not. If it is complete, the transaction is committed and LET entries are cleaned. Otherwise the correspondent LET items are stored into the NRT table for later rolling back. The algorithm follows:

Algorithm 4.

```c
recovery(let_tb, trm_tb, nrt_tb)
LET *let_tb[];
TRM *trm_tb[];
NRT *nrt_tb[];
{
    if (c(p_i, d_j) : T & no trm_tb.hn = j)
        there is a trm_tb.data = "no"
        /* the returned message is not complete or there is
            "no" returns from other relevant hosts */
        /* abort the transaction */
        abort_t(let_tb, nrt_tb);
    else
        commit the transaction;
        cleanup let_tb;
}
```

4. Properties

Following properties can be established for our R2 transaction management model. We outline informal proofs here.

Property 3.1. If every relevant host reports "yes" to the transaction, then there is no host failure during the transaction processi.

Proof: A relevant host reports "yes" if and only if it participates in the transaction. That means every host will report "yes". If a relevant host reports "yes" in step 11 of Algorithm 2, it is committed. We have assumed reliable message delivery, so these "yes" messages will certainly arrive other relevant hosts. So, every relevant host will finally reach step 13 of Algorithm 2: commit transaction.

Property 3.2. If any relevant host reports "no" to the transaction, then there is no host failure during the transaction processi.

Proof: A relevant host reports "no" if and only if it does not participate in the transaction. For the relevant host which reports "no" in step 8 of Algorithm 2: receive "no" to all other relevant hosts and abort the transaction. This report will reach all other relevant hosts. So, a relevant host will finally reach step 14 of Algorithm 2: commit transaction.

Property 3.3. If any relevant host fails, the transaction may commit or abort.

Proof: As we have a reliable delivery network, step 5 of Algorithm 2 will deliver all the "yes" message to a relevant host and step 12 will receive messages from the relevant hosts. They will be viewed as unbreakable transactions. Let i_1, ..., i_m be relevant hosts of transaction and k = i_j, 1 < j < m.

If host k fails within step 1-6 of Algorithm 2, it will do nothing during its recovery. Other hosts will receive step 11 or 12 of Algorithm 2. As k fails, these hosts have timeout during reporting / waiting and cannot do or receive from k. So they will finally abort the transaction.

If host k fails within step 7-11 of Algorithm 2, it will abort the transaction during recovery because of the first condition of the `else` statement in Algorithm 2. This transaction may commit during recovery if k is the only relevant host and step 10 of Algorithm 2 is performed. Other relevant hosts will also abort the transaction because of the timeout.

If host k fails within step 12 of Algorithm 2, all the relevant hosts will be stored in TRM table. During recovery the host will abort the transaction if there are any returns or will commit the transaction if all the relevant hosts return "yes."

Similarly, if host k fails in step 13 of Algorithm 2, the host will commit the transaction during recovery. If
k fails in step 14 of Algorithm 2, the host will abort the transaction during recovery.

Property 3.4. If a transaction commits, all its RPCs are performed. If a transaction aborts, all the performed RPCs of the transaction will be rolled back.

Proof: A transaction commits if and only if all the relevant hosts are willing to commit. That means all the RPCs are performed by these hosts. When a transaction is aborted, all the performed RPCs of a relevant host are put into the NRT table by abort_t() function. This table is then processed by the function rollback_nrt() periodically.

Maintaining transaction atomicity is not a cheap work. The model has the following overhead:

1. Three stable storage tables are used to denote the transaction processing states. They are crucial during host recovery.
2. Since every relevant host of the transaction must be notified of each other’s decision, it is easy to know that if there are m RPCs in a transaction call, then at most m(m-1) messages will be used to commit the transaction.
3. Function rollback_nrt() is called periodically to do the rolling back operations.

5. Remarks

A model for decentralized management of RPC transactions in a distributed system is presented. After the introduction of the problem and the RPC transaction model, we described the algorithms and some properties of the transaction management model. The decentralized RPC transaction management model has no transaction coordinators and therefore can tolerate any host’s failure.

Our model is transparent to programmers. It can act as a run-time system within the programming environment. Programmers will not have too much burden to maintain the RPC transactions in their programs. They can use RPC transaction calls as usual RPC calls and the system will do all the job. We feel this is better than the language level implementation.

The implementation of the system is under way. We use our centralized version as a basis. Currently we use disk space as the stable storage. So the three tables (LET, TRM, and NRT) are stored as disk files. This causes a longer time for logging the table entries (about 50 ms per entry on average).

References

The Design and Implementation of a Distributed Program Monitor

Wanlei Zhou

Department of Information Systems and Computer Science, National University of Singapore

One of the reasons that debugging distributed programs is much more difficult than sequential programs is the communication among processes. The ability to provide for communication events (as well as other events) as they happen during program execution is fundamental to any debugging tool. Most articles about distributed debugging and monitoring are message passing oriented. As the remote procedure call (RPC) method becomes more popular, the need to debug RPC-oriented programs increases. This article presents the design and preliminary implementation of an RPC-oriented program monitor that can record all events of an RPC-oriented program’s execution in the monitor’s database. Facilities are provided for programmers to define, choose, and combine events that will be recorded. Partial ordering among events is built after the program’s execution. A user can use this relation to trace and replay the program’s execution. The monitor has been tested on networks consisting of Apollo/Sun/Digital Equipment workstations.

1. INTRODUCTION

Debugging a distributed program is usually very difficult. The main reasons for this difficulty are the process concurrency and nondeterminism intrinsic to distributed programs. In a distributed program, many events may happen simultaneously; usually, we cannot completely order events during execution. During debugging, we must comprehend the concurrent execution of a number of components, a very difficult task for the human intellect. Also, repeated execution of a distributed program can result in different communication and cooperation patterns between the concurrent components of the program, making identification and repair of bugs more difficult.

Sophisticated debugging tools are urgently needed. Unfortunately, few distributed debuggers are available today to support the development of distributed applications [1], especially in the area of remote procedure call (RPC)-oriented distributed programming. Most existing distributed debugging tools and models are based on the message-passing model of process interaction, in which the send and receive primitives are used in communications. RPC-oriented programs, which are our main concern here, have a different view of process interaction [2]. In these programs, when an RPC is issued, we want to know how the relevant server responds to the call and how the returned results are accepted by the caller. That is, issuing an RPC, receiving and processing the call, returning the call, and receiving the results of the call are the important events in an RPC program. There is no need to investigate how the calling message is sent from the caller to the callee and how the underlying communication primitives are used.

There are two major approaches to the debugging of distributed programs, namely, debugging with repeated execution of the program (or cyclic debugging) [3], and debugging with trace of program execution [4]. In cyclic debugging, a user executes the program in a controlled manner until an error is detected. The program can be reexecuted to produce the same execution behavior. This is a very convenient way to debug small programs or programs that have little interaction among their concurrent parts. For a larger distributed program, however, executing the program several times while repeatedly setting breakpoints may be very costly. Also, sometimes the reexecution of a distributed program may not result the same behavior because of the nondeterministic characteristics of the program. In debugging with trace, no reproducible behavior is needed. The generated trace is determinis-
From the text, it seems that the document is discussing advanced debugging techniques and their implementation. The text mentions a method for debugging called EBBA (Event-Based Distributed Debugging Abstraction) and describes how it can be used to monitor and trace the execution of programs.

The text also touches on the concept of compound events and how they can be used to improve debugging efficiency. The authors mention the use of Petri nets and replay tools as ways to improve debugging processes.

Furthermore, the text mentions the importance of reexecution of programs and the need for hardware and software support. It highlights the challenges of debugging distributed systems and the need for efficient monitoring and tracing techniques.
[12]. We believe our definition of events has advantages in the expression of concurrent events, especially RPC calls. Our debugger can monitor any RPC program irrespective of whether it commences execution before the debugger or not (for example, a server program usually executes "forever"). The only requirement is that the program has to be linked with a debugging library. After debugging, the program can be relinked with ordinary libraries for better performance. Also, no operating system modification is involved, and some techniques are used to reduce the interference of the monitor to a monitored RPC program. For example, we use a debugging library to provide replacements for some operating system and RPC service calls. The main thing a replacement call does, except for maintaining normal operation, is to create an event and queue the event to a local event queue. The work requires only local procedure calls and is quite fast. Our monitor costs an execution program less than 5 ms (on average) to form a primitive event and insert it into the event queue. After that, the execution program continues simultaneously with the event log-in process. This is an improvement over the EBBBA system [6], where the cost is 10-20 ms. Execution trace of an RPC program is stored in a distributed data base; proper methods are used to replay the execution by using the information in the data base. Our replay facility is better than that of LeBlanc and Robbins [8] in that ours allows simultaneous display of events of several concurrent processes. The preliminary version of the monitor has been running on networks consisting of Apollo, Sun, and Digital Equipment workstations.

We use Apollo/HP's network computing system (NCS) [14, 15] in our research. NCS is a commercial RPC system which runs on Apollo, Sun, and Digital Equipment workstations. In this system, a distributed program (here, an RPC-oriented distributed program, or in short, an RPC program) consists of several program parts (PPs). Each PP can be a server or a client program (or both) which can be executed on any host in the network. We assume that these PPs communicate with each other by using only RPCs. A PP can fork to several processes if necessary. When debugging an RPC program, one has to monitor the execution of all its PPs located on several hosts.

2. DEFINITIONS AND ASSERTIONS

We define two sets which will be used in the following description. We use $\Sigma$ to denote all possible events of a distributed program. Usually, a portion of the events in $\Sigma$ may occur during an execution of the program. We use $E$ to denote all the events occurring in a particular execution of the program. That is, we usually have $E \subseteq \Sigma$.

2.1 Primitive Event

To monitor events, we have to assign each event a unique name.

**Definition 1.** An event name is a text object with the following syntax:

eventname := TimeStamp . SeqNo . HostID[ . suffix]

The terminals have the following meaning:

- **TimeStamp** is the value of the local clock register when the event occurred.
- **HostID** is an identifier for the host on which the event occurred.
- **SeqNo** is a cardinal value (called sequential number) used to distinguish events on the same host with the same time stamp.
- **Suffix** is a programmer-defined string used to characterize one or more event attributes. It is optional.

If $en$ is an event name, we use $en.t$ to denote the time stamp, $en.s$ to denote the sequential number, $en.h$ to denote the host, and $en.a$ to denote the suffix component. Because the suffix component is optional, we need to achieve uniqueness with the first three components. Clearly, $en.h$ can be used to identify on which host the event happened and $en.t$ denotes the occurrence time (relative to the local host) of the event. The use of the local clock register necessarily partitions (real) time into discrete intervals, as indicated in Figure 1. Here, events $a$ and $b$ have the same time stamp $t_i$.

![Figure 1. Events with the same time stamp value.](image-url)
In the case that \( n \) events happen within one time stamp interval at the same host as above, \( \text{en.r} \) is used to distinguish these \( n \) events. We maintain an integer variable \( \text{seq} \) (called sequential number variable) at each host. The variable is exclusively accessed by the \text{TimeStamp} mechanism of the event generator, as indicated by (in C notation):

\[
a.r. = (\text{seq} + \text{counter}) \mod N.
\]

The accessing of \( \text{seq} \), and its incrementing, is regarded as an atomic action. So if \( b \) happens at the same interval as \( a \) but the event generator accesses the sequential number variable after event \( a \), then \( a.r \) and \( b.r \) will have different values. We take \( N = 255 \), which gives 256 distinct values for the \( \text{en.r} \) field. These sequential numbers cannot be directly used to order the events in the same time stamp interval. For example, event \( a \) may be assigned to value 255 while \( b \) is assigned to 0. We will discuss this in section 4.1. The assignment and usage of the \text{affix component} will be described in section 3.5.

Another possible way to obtain unique event names is to let the \( \text{seq} \) be a sequential number register (say, 32 bit) and use its sequential values as event identifiers. In this case, the time stamp is unnecessary. This approach has two shortcomings. First, the sequential numbers it generates are still limited. That is, if there are too many events to be named, this approach may not achieve naming uniqueness. Second, by using time stamps, one may have infer the time between two events; this may give some hints for debugging. But by using the sequential number register, it is impossible to make that inference. So we think our approach is the right choice.

**Assertion 1.** In the above scheme, if the maximum number \( n \) of events within a time stamp interval in any host satisfies \( n \leq N + 1 \), then all events in a distributed program are uniquely identified by using their event names.

**Proof.** It is evident that if two events of a distributed program occur in different hosts, then they will have different event names because of the \text{HostID} component of the event name. It is a basic assumption that host identifiers are unique through the distributed computer system. It is equally evident that if two events have a different time stamp, they will have different names. So we need only consider the case where \( n \) events occur in one host during the same time stamp interval. In that case, the host identifiers and time stamps of all these events are the same. The only way to distinguish them is through the sequential numbers.
Definition 3. Let $E = (e)$ be the set of all events of an RPC program's execution. For $x, y: E$ we say $x$ is the predecessor of $y$ if and only if $x$ and $y$ satisfy one of the following conditions:

1. $x \alpha y$ (in different processes)
2. $x$ happened before $y$ within the same process
3. $x$ and $y$ are in different processes and there exist events $a$ and $b$ such that $a \alpha b$ and $x$ is a predecessor of $a$ and $b$ is a predecessor of $y$
4. $x = y$

We denote this as $x \preceq y$. We also say that $y$ is a successor of $x$. Especially, if $x \alpha y$, we say there is one remote relation between them, and call $x$ a remote predecessor of $y$ and $y$ a remote successor of $x$.

Assertion 2. The pair $(E, \preceq)$ is a partially ordered set.

Proof: A set is partially ordered if it is reflexive, antisymmetric, and transitive (16). Because we have defined $x \preceq x$ for any $x \in E$ (condition 4 of definition 3), it is reflexive. Suppose $x \preceq y$ and $y \preceq z$, $x, y$ and $z \in E$. If $x, y,$ and $z$ are in the same process, then by condition 2 of definition 3, we have $x \preceq z$. If $x, y; P_1$ while $z; P_2$ (where $P_1$ and $P_2$ are event sets of different processes of a distributed program), then by condition 3 of definition 3, there exist events $a$ and $b$ such that $y \preceq a$, $a \alpha b$, and $b \preceq z$.

(1)

If there is only one remote relation between $y$ and $z$, then we must have $a: P_1$ and $b: P_2$. So, by condition 2, we have $x \preceq a$, $a \alpha b$, and $b \preceq z$. That is $x \preceq z$. Now, suppose the assertion holds for $n$ remote relations. If there are $n + 1$ remote relations between $y$ and $z$ when equation 1 holds, then the number of remote relations between $y$ and $a$ (or equally, between $x$ and $a$) must be less than or equal to $n$, and so must the number of remote relations between $b$ and $z$. So we still have $x \preceq z$. By induction, the assertion holds. The proof for $x; P_1$ while $y; z; P_2$, and $x, y,$ and $z$ are in three different processes is similar; so the pair $(E, \preceq)$ is transitive. The antisymmetry can be proved by enumerating conditions of definition 3. Let $x \preceq y$ and $y \preceq x$. Because $x \preceq y$, $x$ and $y$ must satisfy one of the conditions in definition 3. Suppose they satisfy condition 1 of definition 3, that is, $x \preceq y$. Then $y \preceq x$ must be false, because, by corollary 1, $y \preceq x$ is false and also $x$ and $y$ cannot satisfy other conditions. Suppose $x$ and $y$ satisfy condition 2 of definition 3.

Then they are in the same process and $x$ occurred before $y$. That means $y \preceq x$ is false. Condition 3 is actually an extension of conditions 1 and 2. So $x$ and $y$ cannot also satisfy condition 3. The only way for both $x \preceq y$ and $y \preceq x$ to hold is if $x$ and $y$ satisfy condition 4 of definition 3. That is, $x = y$. So $(E, \preceq)$ is a partially ordered set.

Lamport [17] defined a "happened before" relation to describe events in a distributed system. The message-passing model's send and receive are used in his definition. The definition provides a general model for ordering events of a distributed program. Here we are interested in RPC events, so we defined our predecessor and successor relations using the RPC communication model and included a fact part in the event definition.

2.2 Combined Event

Sometimes a user may be interested in the combination of several events. For example, if a server has two remote procedures that will access an object, it is interesting to see if these two procedures are all called during the execution, or to know their execution order. To define events combination, we borrowed a notation from temporal logic [18] and define as follows:

$\langle A \rangle$: Eventually operator. There is a time point after the reference point (the present time) at which $A$ occurs.

The following definitions combine several events to form a new event.

Definition 4. Consider events $e_1, e_2: \Sigma$, where $e_1 = (f_1, m_1)$ and $e_2 = (f_2, m_2)$. By $f_1 \cdot f_2$ we mean that the fact part of $e_1$ is followed by the fact part of $e_2$. By $f_1 + f_2$ we mean that the fact part of $e_1$ and the fact part of $e_2$ are independent. Similarly, we define $m_1 \cdot m_2$ as message $m_1$ followed by message $m_2$ and define $m_1 + m_2$ to mean they are independent each other. The definition is also valid for more than two facts (messages). But in our monitor, we have only implemented binary operations (see section 3.5).

We do not give the details of followed-by and independent operations here. They will be discussed in section 3.3, where we will define the actual data structure of an event and describe these two operations. In the followed-by operation, two or more facts (or equally, messages) are sequentially connected, while in the independent operation, there is no such connection between these facts (messages).
Definition 5. Let $e_1, e_2 \in \Sigma$ (they do not necessarily occur during one execution of a distributed program), we define:

\[ e_1 + e_2 \quad \text{if} \quad (\Diamond e_1 \text{ and } \Diamond e_2) \quad \text{and} \quad e_1 \leq e_2. \]  
\[ e_1 \& e_2 \quad \text{if} \quad (\Box e_1 \text{ and } \Box e_2) \quad \text{and} \quad \neg (e_1 \leq e_2). \]  
\[ e_1 \lor e_2 \quad \text{if} \quad (\Box e_1 \lor \Box e_2). \]  
\[ e_1 \land e_2 \quad \text{if} \quad (\Diamond e_1 \lor \Diamond e_2). \]  
\[ e_1 \lor e_2 \quad \text{if} \quad (\Diamond e_1 \lor \Diamond e_2). \]

So, formula 2 means that both events occurred and they have a less-than-or-equal-to relationship. Formula 3 means that both events occurred, but there is no relationship between them. Formula 4 simply means that the two events occurred. Finally, formula 5 means that either one or both of the events occurred.

It is easy to see that

\[ (e_1 \lor e_2) \rightarrow (e_1 \land e_2) \rightarrow (e_1 + e_2) \rightarrow (e_1 \& e_2). \]

\[ e_1 \land e_2 = \begin{cases} e_1 + e_2 & \text{if } e_1 \leq e_2 \\ e_2 + e_1 & \text{if } e_2 \leq e_1 \\ e_1 \lor e_2 & \text{if } e_1 \leq e_2 \text{ and } e_1 \lor e_2 = e_1 \land e_2 \text{ or } e_1 \text{ or } e_2. \end{cases} \]

Definition 6. If $e_1 = (f_1, m_1)$ and $e_2 = (f_2, m_2)$ are events, then

\[ e_1 \lor e_2 = (f_1 \lor f_2, m_1 \lor m_2), \]
\[ e_1 \land e_2 = (f_1 \land f_2, m_1 \land m_2), \]
\[ e_1 \land e_2 = (f_1 \lor f_2, m_1 \lor m_2). \]

are also events, where

\[ f_a = \begin{cases} f_1 \lor f_2 & \text{if } e_1 \leq e_2 \\ f_1 \lor f_2 & \text{if } e_2 \leq e_1. \end{cases} \]
\[ m_a = \begin{cases} m_1 \lor m_2 & \text{if } e_1 \leq e_2 \\ m_1 \lor m_2 & \text{if } e_2 \leq e_1. \end{cases} \]

We define these new events as combined events and define $e_1$ and $e_2$ (or $e_1$ or $e_2$, depending if both of them or only one of them occurred) as their predecessors (also called components of the combined event). Combined events are usually defined by programmers, so they are of interest to users. For example, in Figure 2, there are two processes $P_1$ and $P_2$. The following relations among their events hold (of course, there other relationships exist):

\[ a_1 + b_1, a_1 + b_2, a_1 + b_3, a_1 + b_5, \text{ and } b_1 * a_3. \]

It is easy to see that

\[ e_1 \lor e_2 = e_2 \lor e_1; \quad e_1 \land e_2 = e_2 \land e_1; \]
\[ e_1 + e_2 = e_2 + e_1. \]

The priority of the above operators are, from high to low, $\lor$, $\land$, $\star$, $\ast$, so the expression $e_1 + e_2 \land e_3 \lor e_4 + e_5$ is actually $(e_1 \lor (e_2 \land e_3)) \lor e_4 + e_5$.

Assertion 3. If $E$ is the set of all (primitive and combined) events of an RFC program's execution, then $(E, \leq)$ is still a partial ordered set.

Proof. From the above definitions, it is easy to see that $(E, \leq)$ is still reflexive, asymmetric, and transitive.

Definition 7. We call event $a$ the immediate predecessor of event $b$ if

1. $a \leq b$, $a$ is not $b$, and if $c \leq b$, then $c \leq a$, or
2. $a$ is a component of a combined event $b$.

For example, in Figure 2, $a_2$ and $b_1$ are immediate predecessors of $a_3$ but $a_1$ and $b_2$ are not.

3. MONITOR STRUCTURE

3.1 Overview

The monitor consists of a controller and a group of managing servers. The controller has two main parts: a user interface (which incorporates a command interpreter and input/output functions) and a filter. A managing server consists of a server (MS server), an event queue, and an event data base. Each host that which supports one or more monitored program

![Figure 2. Events relations.](image-url)
parts has a managing server. The controller can be located at any host. By communicating with the associated servers, the controller can present the monitored results to the user. Figure 3 illustrates the structure of the monitor.

Debugging using this monitor system involves three stages or steps.

1. Monitoring. In the monitoring step, all events that occur on a particular host are monitored by the local MS and recorded in the local event database. All primitive events including RPCs, executions, and process forks in both client and server program parts are monitored. Furthermore, a user can define combined events via the event definition file (EDF); described in section 3.5. The monitor system is able to record such combined events.

2. Ordering. After all the events are recorded, the programmer uses the ordering step to order events. At this time, each MS exchanges remote predecessor/successor information through the controller and has all remote relationships ordered. Then, local predecessor/successor relationships are established by each MS over its local event data base.

3. Replaying. By combining the results on all related event data bases, the filter presents an execution trace of the distributed program to the user. The displaying speed and viewing contents are controlled by the user through the command interpreter. Two kinds of user interfaces are provided, namely, a command-level interface and a graphics interface. A user can use commands to order the monitor to start a remote/local program part and leave the monitor to monitor that part. On the other hand, the user can monitor an executing program part.

3.2 Debugging Library

There is no doubt that an effective distributed debugger has to be deeply embedded into the operating system to achieve sufficient speed and transparency. To monitor a program’s activity without causing any side effects in its behavior, operating system kernel modification or hardware support is essential. Because of the difficulty of modifying operating systems and providing hardware support, most of debugger and monitor researchers use software techniques as the substitute. This makes the implementation much easier at the cost of efficiency, especially for real-time systems. Performance may be completely unacceptable for real-time programs. At this stage, we are not in a position to create hardware support or change the operating system kernel. Instead, we have provided a debugging library, which has to be linked with the program that is to be monitored. This library provides replacements for some of the operating system calls, such as fork. It also has some functions that replace NCS-related calls and other service functions. Each replaced function does the following two things:

![Image](image_url)

*Figure 3. The structure of the distributed monitor.*
1. Event creation and queueing. It first forms the primitive event, then inserts the event entry into the local event queue for the local MS to process.

2. Normal execution. This does the normal work of the original system/NCS call.

The first step requires only local procedure calls and is quite fast. On average, the time for event creation and queueing is less than 5 ms on Apollo and Sun workstations. The monitored program part resumes normal execution after this step. After the programmer judges that the program has been debugged, the program can be relinked with ordinary libraries.

3.3 Managing Server

On each host, there is an MS which consists of a server (MS server) and an event data base. Each event data base is simply a set of entries where each entry records an event and has the structure indicated below.

```c
typedef struct msg {
    char *ty;    /* type info */
    byte *m;     /* message info */
} MSG;

typedef struct event {    /* event info */
    char *name;      /* event name */
    int pid;         /* process ID number */
    char *prg_name;  /* program part name */
    char *p_name[MXPRED]; /* predecessor event names */
    char *s_name[MXSUC]; /* successor event names */
    char *next;      /* next event name */
    char *fact_info; /* fact information */
    MSG *message;   /* message part of the event */
} EVENT;
```

The name, pid, prg_name, p_name, and s_name fields are self-explanatory. The fact information (fact_info) of an event is a character string that provides a readable description of the fact. For primitive events, they are assigned by the debugging library. For example, they may be "begin RPC call (RPC name)" or "fork new process." For combined events, they are assigned by programmers (see section 3.5). The message part of an event is stored as a string of bytes and a piece of type information. The filter uses the type information to interpret the byte string and display the result to the user through the command interpreter.

The next field of an event is used to record the result of a followed-by operation. If event \( x = y \circ z \), then after the ordering step, we will have \( x \rightarrow \text{next} = y \rightarrow \text{name} \) and \( y \rightarrow \text{next} = z \rightarrow \text{name} \). For other cases, this field is left empty.

If \( x = y \circ z \), the fact and message information of \( x \)'s components can then be accessed by following the \( x \rightarrow \text{next} \) field. If \( x = y + z \), then \( x \)'s fact and message information can then be accessed by using the \( x \rightarrow \text{p_name} \) field, because any component of \( x \) is a predecessor of \( x \). There is no predefined accessing sequence (as in + operation) for these components.

Now we can have some concrete understanding of what an event is represented in our monitor. For an event \( e = (f, m) \), we can view \( e.m \) as the message field of its event representation; others of the representation belong to \( e \)'s fact part.

An MS server has the following functions:

1. Data base management. It is responsible for the management of the local event data base. The data base is protected by the MS server; any access to it must go through the MS server.

2. Event logging. When an event occurs, the event name is built and is put into a local event queue by the executing program (see section 3.2). The local MS server is then responsible for logging it in the local event data base. In this case, the monitored process can continue immediately after the event has been queued, instead of waiting for the local MS server to insert the event into the data base.

3. Communicating with the controller. All communications among servers are conducted by the controller. Many commands (see section 3.4) are issued through the controller and performed by the server. This is the only way a user can access the event data base.

In the second function above, if there are too many events and the MS works too slowly (for
example, the time required to monitor a PP or the congestion of disk accessing), the event queue may become full. In that case, we let the execution program store the event in to a temporary file which the server will check when the monitoring step is over. Then all events in the file will be inserted into the local event data base and the file will be deleted. Storing an event into the temporary file is a relatively long process—it will cost the program about 300 ms on average. Fortunately, the maximum length of the event queue is defined large enough to hold all events of an ordinary program part of a distributed program, so in many cases, the temporary file will not be used.

3.4 Controller

The controller consists of a command interpreter and a filter. A list of main user commands follows (Table 1). Under some commands, there are several subcommands. For example, in the "Replay the execution of a process" command, the monitored events are displayed one by one according to the partial ordering (in the case of several successors, the first is chosen arbitrarily). Each "picture" is displayed for 5 s. The user now can have subcommands such as "interrupt and replay," "change display speed," "single step" (the user is responsible for choosing the predecessor or successor event to display), "view the message part of an event," and "continue auto-replay." As another example, when invoking a program part for monitoring, the user can specify through the subcommands the class of primitive events not to be monitored, and whether the message part of the events is to be recorded (the default is to monitor all primitive events and their message parts). This may help the user to select and store events that may be interesting while not using too much memory. It is particularly useful for debugging large programs.

```
EDF ::= BEGIN
     <SPEC>
     <COMBINE>
END

<SPEC> ::= <PresSpec> ; <ComSpec> ;

<PresSpec> ::= Primitive Event:
              Event_Affix { , Event_Affix }

<ComSpec> ::= Combined Event: (variable, fact_info)
             { , (variable, fact_info) }

<COMBINE> ::= <EventAssignment> { ; <EventAssignment> }

<EventAssignment> ::= variable <EventExpression>
<EventExpression> ::= <FFollow> | <EventExpression> | <FFollow>

<FFollow> ::= <FUnion> | <FFollow> * <FUnion>

<FUnion> ::= <FInter> | <FUnion> union <FInter>

<FInter> ::= <Operand> | <FInter> inter <Operand>

<Operand> ::= Event_Affix | variable | (<EventExpression>)
```

In the command-level interface, all commands are provided in menu-driven format and only one process can be replayed at a time. If the replay is carried out in the graphics interface, the user can open several windows to view the concurrency of different processes and program parts.

The filter has three main functions. First, it maintains the communication between the command interpreter and the MS server. After the command interpreter has accepted and interpreted a command, it is passed to the filter to have the appropriate MS servers execute the command. The results of the execution are then interpreted and passed to the command interpreter through the filter. Second, the filter maintains the communication between MS servers. If an MS server wishes to communicate with another MS server, it first communicates with the filter, then the filter communicates with the destination MS server and returns the result to the first MS server. In this case, the programming of an MS server is much simpler. The last function of the filter is to interpret the message part of an event. As mentioned earlier, the message part of an event is stored as a string of bytes and type information during the monitoring step. When a user requests to view the message part during replay, the filter will find the appropriate message and use the type information to interpret the byte string, then pass the result to the command interpreter. At this point only simple types can be interpreted by the filter.

3.5 Event Definition File

All primitive events are automatically logged by the monitor if the monitored program is linked with the debugging library. Sometimes a user may find it more convenient to define some high-level events and use them in debugging. An EDF is used for that purpose. The syntax of the EDF specification language is indicated below.
Table 1. Controller Commands

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Use EDF</td>
</tr>
<tr>
<td>2</td>
<td>Invoke PP for monitoring</td>
</tr>
<tr>
<td>3</td>
<td>Monitor an executing server PP</td>
</tr>
<tr>
<td>Ordering</td>
<td>Order events of a process</td>
</tr>
<tr>
<td>2</td>
<td>Order events of a PP</td>
</tr>
<tr>
<td>3</td>
<td>Order combined events</td>
</tr>
<tr>
<td>Replaying</td>
<td>Replay execution of a PP</td>
</tr>
<tr>
<td>2</td>
<td>View combined events</td>
</tr>
<tr>
<td>3</td>
<td>Dump trace of a PP</td>
</tr>
<tr>
<td>5</td>
<td>Dump combined events</td>
</tr>
<tr>
<td>6</td>
<td>Clean up trace of a PP</td>
</tr>
</tbody>
</table>

The following notes are relevant:

1. **Variable** has usual meaning as in the C programming language.
2. **Event_Affix** is the same as affix, defined in definition 1 (a character string).
3. Any variable that appears in the righthand side of an **EventExpression** must be the lefthand side of an earlier **EventExpression**.
4. **Fact_info** is a character string as described in section 3.3.
5. **Inter** is the keyword for operation ∩ and union is the keyword for operation ∪. They are defined in definitions 4–6.

We give a very simple EDF file below. It defines a combined event **BothAccessed** as the intersection of two primitive events **AccessingDB1** and **AccessingDB2**. That is, if both primitive events occur, then the combined event is also deemed to occur.

```plaintext
BEGIN
  primitive Event:
  AccessingDB1, AccessingDB2;
  Combined Event:
  (BothAccessed, 'Both DBs are accessed');
  BothAccessed = AccessingDB1 inter AccessingDB2;
END
```

Several steps are needed to use an EDF file. First, the user inserts affix definition functions into the monitored program parts before each primitive event to be used in the EDF file. The format of the affix definition function (defined in the debugging library) is

```plaintext
affix_define(affix),
```

where affix can be any character string (such as **AccessingDB1** and **AccessingDB2** above). When executed, this function will set an affix flag to true. The primitive event occurring next will then detect this flag and will use the affix to build the event name.

Second, the controller reads the EDF file for the distributed program. It then builds an evaluation table according to the definitions in the EDF file.

Third, when any of the affixed primitive events occur, they are sent to the controller by the local MS servers (of course, the local MS servers also record them as usual). The controller then evaluates the combined event expressions expressed in the evaluation table by using the affix part of the affixed primitive event. If any of the expressions are true, the combined event is then recorded into the event data base. Then the evaluation take place once more, in case the combined event is also a component of another combined event. The predecessors of a combined event are all the events (primitive and/or combined) in the righthand of the event expression. For example, combined event **BothAccessed** has two predecessors, **AccessingDB1** and **AccessingDB2**. If no combined event expression is true, the affixed event is then stored into an evaluate structure maintained by the controller, where it waits to be evaluated again if other affixed primitive events occur.

We have made some restrictions on the implementation of the EDF specification language. First, a combined event can only contain two events components. That is, only the following four kinds of combined events can be defined:

```plaintext
x = y * z, \hspace{1cm} x = y + z, \hspace{1cm} x = y \land z, \hspace{1cm} x = y \lor z.
```

In this case, the interpreting of the EDF file is not complex. Second, only one EDF file can be defined for a distributed program and at most 20 combined events can be defined in an EDF file. This avoids evaluation table explosion.

4. **TRACE ANALYSIS**

4.1 **Ordering Events**

As we have pointed out, the local clocks of the hosts in the distributed system are not synchronized, so we can not use the time stamp from definition 1 to order all events. According to assertions 2 and 3, there does exist some partial ordering among all events occurring during an execution. The following steps are used to establish the partial ordering among all events:

1. **RPC-related predecessors**. When an RPC-related event happens (for example, the issuing or ending of
an RPC), it will cause the occurrence of an event which belongs to another process (and possibly, on another host). In that case, the first event is changed (by the debugging library) to carry not only the original information, but also the event’s name. On the other hand, the second event is also changed (by the debugging library) to not only receive the original information, but also the first event’s name, and this name is stored by the local MS server as the immediate predecessor of the second event. All RPC-related predecessor events can be stored in this way.

2. Process fork-related predecessors. When a process fork event occurs, it will cause a new process to be set up and executed. This event is changed (by the debugging library) to carry the name of the event; the first event of the new process will use the carried name as its immediate predecessor event.

3. Combined event-related predecessors. When an event with an affix definition occurs, it will cause the controller to evaluate the related combined event expressions as described in section 3.5. So, the name of the event is sent to the controller and stored as one of the immediate predecessors of the related combined events.

4. Form all remote successors. In steps 1-3, all remote predecessors will be established after the termination of the monitoring step. The remote successors are built by each involved MS server and the controller at this moment. Each MS server checks all events in its local data base. If an event \( x \) has a remote predecessor named \( y \), then the MS server will be responsible of storing \( x \) as a successor of \( y \).

It is easy if \( x \) and \( y \) are in the same data base (for example, the fork events). Otherwise, \( y \cdot h \) is used to locate the MS server to which it belongs; \( y \)'s successor will be stored by the communication of these two MS servers through the filter.

5. Form all other successors and predecessors. All the events within a process are ordered by their time stamps. If two such events have the same time stamps, then they are ordered by their sequential numbers. That is, we view all sequential numbers (from 0-255) as having round ordering such that

\[
0 < 1 < 2 < \cdots < 254 < 255 < 0 < 1 \cdots ,
\]

As we have assumed that no more than 256 events occur between two time stamps, building the ordering among these events is straightforward. So, in one process, the immediate predecessor of event \( y \) is event \( x \) if \( x \cdot t \) is immediately less than \( y \cdot t \) and the (immediate) successor of \( y \) is event \( z \) if \( z \cdot t \) is immediately greater than \( y \cdot t \). If \( e_1, e_2, \ldots, e_n \) are all events that have the same time stamp \( t \) \((n < = 256)\), then the round ordering equation (6) is used to order them. This ordering process is performed by each MS server concurrently.

We use the example of Figure 4 to illustrate the essential aspects of these steps. The figure shows two processes \( A \) and \( B \). They execute on different hosts and their local MS servers are indicated by \( MS_A \) and \( MS_B \), respectively. Suppose that \( A \) makes an RPC to one of \( B \)'s procedures. Let us use \( a_1 \) to represent the event “issuing an RPC call” and \( a_2 \) the event “RPC return” for \( A \). For \( B \), we use \( b_1 \) to represent the event “begin execution of the remote procedure” and \( b_2 \) the event “completion of remote procedure execution.” When \( a_1 \) occurs, \( a_1 \)'s name is
Our monitor is event based. We believe our event definition has advantages to express events of distributed programs. Particularly, it is better than many existing methods in expressing RPC-oriented events such as RPC calling and RPC execution. Our debugging library, the local MS servers, the distributed event queues, and data bases provide an efficient way of reducing the interference of the monitor with the monitored RPC programs.

ACKNOWLEDGMENT

I thank Dr. Brian Molinari for reading this manuscript and for his comments during the research. I also thank the reviewers for their helpful comments and suggestions.

REFERENCES


A Layered Distributed Program Debugger

Wanlei Zhou
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 0511

Abstract

Debugging distributed programs is much more difficult than debugging sequential programs. One of the reasons is the communication among programs (processes) which may happen concurrently and nondeterministically. To be able to analyze such communication events is an essential task for any distributed program debugger. This paper describes the design and preliminary implementation of a layered distributed program debugger. The debugger helps a user to locate bugs, to analyze a distributed program and to fix bugs.

1 Introduction

Debugging is a process of isolating, diagnosing and correcting program errors. This is one of the most costly phases in the program development process in large systems[2]. Debugging a distributed program is usually much more difficult than debugging a sequential program. A distributed program can be viewed as a set of program parts that work together on a single task, and the concurrency and communication among these parts are the main reasons that make debugging of distributed programs difficult[3]. To be able to analyze such concurrent and communicating events is therefore an essential task for any distributed program debugger.

Debugging a distributed program can be divided into two phases. At the first phase, called localization, we need to locate which part of the program has a bug. Then at the second phase, called analysis/fixing, we analyze the code that may cause the bug and fix it. Unfortunately, almost all existing distributed debuggers only provide communication events occurred in lower level. They assist in fixing a bug after having obtained a rough idea about the bug’s localization. This only provides information for the second debugging phase. To dig out the possible bug locations using the existing debuggers is a difficult job as there are usually many events involved.

An ongoing project of high-level debugger for parallel programs is described by Caets et al [1]. They use several abstraction levels (from the coarse-grain interacting processes or threads to textual representation of the program) in their debugger. A top-down method following the abstraction levels is used to locate a possible bug. But the debugger is limited to message-passing on shared memory systems.

This paper describes the design and preliminary implementation of a layered distributed program debugger. It is a significant improvement of our earlier distributed monitor [4] and its preliminary implementation has been tested on networks consisting of DEC/HP/SUN workstations. The debugger helps a user to locate bugs, to analyze a distributed program and to fix bugs. Three steps are used in a debugging process. At the monitoring step the debugger records all the events of a distributed program into a distributed database. In the ordering step, these recorded events are ordered according to a partial ordering scheme. At the debugging step, a user analyzes a distributed program and locates bugs in four levels. In the highest level, the debugger displays the communication relationships between programs (eg., server and client programs). In the second level, the communication between processes is displayed. In the third level, the communication between events is displayed. The forth level is the lowest level; it displays the relevant statements that carry out the communication. The first three levels help the localization of bugs and the forth level helps the analysis/fixing of bugs.

The paper is organized as following: Section 1 is the introduction, Section 2 briefly describes the structure of the debugger, Section 3 describes issues related to the debugger’s implementation, and Section 4 presents an example to illustrate our layered debugging. The descriptions are very brief. For more detailed descriptions, please refer to [5].

2 Structure of the Debugger

In order to monitor events we have to assign each event a name. Definition 1 is the method which can uniquely name any event.

Definition 1. The name of an event (event name) is a "unique ID" (UID), defined as a string of characters and an optional user attached affix. The format is t.r.h.[affix] where t is the timestamp, which is stamped by the local host; r is a sequential number; h is the host ID; and affix is an optional affix name (a character string) which is defined by a user.

If an event name, we use et to denote the timestamp, er to denote the sequential number, en to denote the host, and en.a to denote the affix of event name, respectively. En.h can be used to identify on which host the event happened, and en.r denotes the occurrence time (relative to the local host) of the event. In the case that n events happened simultaneously at the same host, en.r is used to differentiate these n events. It is not difficult to assure that no n (n is a hexadecimal number and n ≤ 255) adjacent sequential numbers generated are the same. So all events of a distributed program can be uniquely identified by using their event names.
3 Implementation Issues

No doubt an effective distributed debugger has to be deeply embedded into the operating system or even has the help of dedicated hardware components for achieving sufficient speed and transparency. Because of the difficulty of modifying operating systems and obtaining hardware support, most debugger and monitor researchers use software techniques as a substitute. This makes the implementation much easier, but the performance is not very good, especially for real-time systems (which may be completely not suitable for real-time programs). At this stage, we provided a debugging library which has to be linked with a program being debugged. This library provides replacements for some BSD4.3 UNIX operating system calls related interprocess communication.

These replacements first perform some work required by the debugger, such as reporting to the local MS of the event’s happening, and then do the normal work of the original calls. After the programmer thinks the program has been debugged, the program can be re-linked with ordinary libraries.

On each host, there is a managing server (MS) which consists of a server and an event database. An MS has several functions. Firstly, it manages the local event database. The database is protected by the MS and any access of it must go through the MS. Secondly, when an event occurs, it is logged by the local MS into the event database. Because the events may happen very fast (or even concurrently), while the logging of an event requires some amount of time, we use an event queue to queue up all events waiting to be inserted into the database. All events are queued into a event queue after their happening, and the local MS looks at the queue and puts the events into the event database. Thirdly, it communicates with the controller. All communications among MSs are conducted by the controller. A lot of commands are issued by the controller and performed by MSs. This is the only way a user can access the event database.

The controller consists of a command interpreter and a filter. The command interpreter accepts and analyses commands from a user and controls the filter to perform the required functions.

The filter has three main functions. Firstly, it maintains the communication between the command interpreter and MSs. After the command interpreter accepts and interpreted a command, it is passed to the filter to have the appropriate MSs to execute the command. Then the results of the execution are interpreted
and passed to the command interpreter through the filter. Secondly, the filter maintains the communication between the MSSs. In that case the programming of an MSS is much simpler. The last function of the filter is to interpret the message part of an event. When the user requests to view the message part during debugging, the filter will find the appropriate message and use the type information to illustrate the byte string, and then pass the result to the command interpreter.

All preliminary events are automatically logged in by the debugger if the program being debugged is linked with the debugging library. Sometimes a user may find that it is more convenient to define some new events during debugging. The Event definition file (EDF) is used for that purpose.

As we have known, the time system of each host in a LAN is not closely synchronised, so we can not use the timestamps in definition 1 to order all events. But the timestamps can certainly be used to order events happened in one process because they always remain within the same host. That is, events within each process of a distributed program can be fully ordered. But it is impossible to fully order events of different processes. As mentioned earlier, there exist some partial ordering relationship among these communication and process fork events. An ordering algorithm is described in [5]. After applying the algorithm, a partial ordering is established among all the events.

After the monitoring and ordering steps, the user goes into the third step, the debugging. As we mentioned before, the first thing of debugging is to locate the bugs. A top-down view of the program is a suitable way of localising bugs.

At the top level (program level), communication between program parts is displayed. A distributed program may have many program parts. The debugger provides a facility to let the user select program parts for display. Because each event entry, there is a field pgra.name specifies the name of the program part, we use this information and the event ordering to draw the communication diagram between program parts. At this level, no event detail is displayed. The user can have a nice top-view of the program communication.

From the top level, the user may have some idea that which part of the program probably has a bug. So some relevant program parts can be selected and the second level (process level) will display the communication between these selected processes. If nothing can be found at the top level, the user can ask the debugger to display all the processes in the second level. We also use a field pid in every event entry and the event ordering to draw the diagram.

Usually from the second level, some processes can be selected for further investigation. The third level (detail level) displays the events and communications between selected processes (of course, it can also display all the events relations of the program). In this level, the event numbers are used and the user can consult these numbers with the event tables (see below).

From the third level, the bugs will be found and the relevant program segments will be displayed using a text editor in the forth level (text level). The user can analyse the program segment and fix the bug here.

During the displays, the user can view event details in two levels. At the table level, the events of a selected process is displayed in a table form (event table). Then the user can select and event within an event table and ask the debugger to display its details (detail level).

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
<th>Pred</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>Server begins</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>Create a socket</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>a3</td>
<td>Bind to a name</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>a4</td>
<td>Receive request</td>
<td>6917.4</td>
</tr>
<tr>
<td>5</td>
<td>a5</td>
<td>Fork child process</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>a6</td>
<td>Forced exit</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Event Table of Process 6911

Figure 2: Debugging: The Level 1 Picture

4 An Example

We show a simple debugging example to illustrate the debugging process. When teaching "Distributed Computing" course, a student asked me to find out why his exercise program did not work properly. The program was a "Send-and-forward" system. The server acted like a message storage. A user used the client program to send a message (with a receiver's name) to the server. The server kept the message until the addressed receiver (also a client program) asked the server to forward messages.

When the student's program was executed, both the server and client program parts were hung up. It was difficult to guess what was happening inside these two program parts. We then used our debugger to record the events of the program. After the event ordering step, we had the top-level diagram of Figure 2.

From this diagram we know that the client sent the request, the server acknowledged and then the client sent the message. We further used the second level to locate the processes that were responsible for the failed communication. Figure 3 is the process level diagram.

Three processes were involved. For better understanding of the communication we asked the debugger to display event tables for all these three processes. Table 1, Table 2 and Table 3 are these tables. For saving space, we use abbreviated notations for event names, for example, the real expression for event a1 is 5f7121812b3c1.00.0282c22e15000000.
We analysed the program section of possibility 1 using a text editor and found out that inside the message storing function, some value of the client socket address was mistakenly re-assigned. That caused the sendto() call of the server child process to send the acknowledgment to an unknown address. After fixing the bug, the program worked correctly.

## 5 Summary

The design and a preliminary implementation of a distributed program debugger is described in this paper. The debugger has several managing servers which record the events of program parts of their hosts into their local event databases. By using an ordering scheme, all events of a distributed program can be partially ordered, and the event graphs in different levels and the relevant event tables can be built. These event graphs and tables are then used to locate the possible bug positions in a top-down manner. Facilities are also provided to define combined events and to view the details of the events. The preliminary implementation of the monitor has been tested on networks consisting of DEC/HP/SUN workstations.

### References


A Rapid Prototyping System for Distributed Information System Applications

Wanlei Zhou

Department of Information Systems and Computer Science, National University of Singapore, Singapore 0511

1. RAPID PROTOTYPING AND ITS CONTEXT

For the past 20 years or so, software system development has been based on the software life-cycle model (or waterfall model) (Boehm, 1976). This model essentially advocates that software projects should consist of a number of distinct phases (Boar, 1984). These are: specification, design, implementation, testing, operation, and maintenance. This model has been modified by many researchers since its inception. But the central idea remains unchallenged; that is, all variations keep the linear structure, and each phase begins only when the previous phase has been completed.

The life-cycle model works very well when the application is both well understood and supported by previous experience, but in general, it has many deficiencies that are too serious to be ignored (Hekmatpour and Ince, 1988). The reason is that some assumptions made by the life-cycle model are no longer true. The life-cycle model reflects the time period in which it evolved. Dramatic changes since then in the environment of the software process promote a reassessment of the model, and new development models are needed to fit the evolved technology and changed application domains.

Computer-aided rapid prototyping has been suggested as an alternative scheme to overcome the deficiencies of the life-cycle model for the development of information system (IS) applications (Blum, 1982; Boar, 1984; Boehm, 1984; Luqi and Ketabchi, 1988; Luqi and Lee, 1989; Balzer and Gabriel, 1989). In this approach, a range of computer tools is used to help the developers to generate prototype programs. When the proposed distributed IS application is too complex; when the user interface is an important part of the system, when the requirements cannot be completely specified at the beginning, or when there are too many uncertainties about the proposed system, rapid prototyping is a suitable model for development.

Several approaches to rapid prototyping exist (Hekmatpour and Ince, 1988). In throw- it-away prototyping, the prototype system is used for a limited period and is usually used for requirements analysis and specification. After that, it is thrown away. The rapid development of the prototype is the greatest need, while the efficiency of the prototype is of little importance. A second approach is called evolutionary prototyping. Here a system grows and evolves gradually. At first, only those parts of the system that are well understood are developed, and the prototype then evolves as the understanding of the whole system becomes clearer during the use of the first prototype. This is very suitable for gradually introducing a new system into an organization and for coping with the changes that take place within the organization as a result of using the system. This method is the most attractive model for IS applications. It is the prototyping method used in this paper.
In all cases, a prototype must be a “working model” of the proposed IS application. The challenge is then to develop the related software tools that will support the prototyping process. In the following sections, we present several tools that help the development of distributed IS application prototypes.

2. STRUCTURE OF THE PROTOTYPING TOOLS

2.1 Network Computing System

We use the Network Computing System (NCS) (Kong et al., 1990; Zhan et al., 1990) to develop our distributed applications. NCS is a set of software tools for remote procedure call (RPC)-based distributed computing. It is developed by HP/Apollo and is adopted by Open Software Foundation (OSF) as a distributed computing standard. The foundation for this system is the Network Computing Architecture (NCA), which supports situations where both data and execution are distributed across one or more heterogeneous networks. The following components are provided by the NCS to assist the development and execution of the programs related to the NCS:

- remote procedure call runtime library,
- Network Interface Definition Language (NIDL) compiler, and
- location brokers.

The RPC runtime library provides the system calls that enable a local program to execute procedures on remote hosts. The location broker then provides the information of remote (and local, of course) services. The NIDL compiler is a tool for developing NCS applications.

The process of a typical NCS application development may be as follows. At first, the programmer uses the NIDL language to write an interface definition that defines all of the remote service interfaces (procedures). The programmer then compiles this definition using the NIDL compiler. In general, there are four output files for an interface definition, where two of them are client stubs, one is a server stub, and the last one is a header file for the use of both client and server programs. The programmer then builds the server program, which implements the remote interfaces described in the interface definition, and the client program, which makes use of the remote procedures (and other application functions). The format for the remote procedure calls in the client program is defined in the interface definition. Finally, the server program is linked with the server stub and the client program is linked with the client stubs. Now the server program can run on the remote host and the client program running on the local host can execute the remote procedures in the same way as it requests local procedures.

One can notice that we do not mention the location broker above. A small and specific application needs to have no recourse to the location broker because the client program knows where the remote services are located. The location broker is very useful in general, however. Usually, a server program must register all of its services with the location broker. The client program can then find the service through the location broker. After the client finds the location of the service, it then calls the service directly. This is called unbound (or allocated) calling. We use this as the standard calling semantics. Of course, NCS also support other calling semantics, such as bound-to-host and fully bound calls.

A distributed program in NCS can be functionally divided into two parts: the server part and the client part. Each part can be located at any host in the network. Usually, a server part manages an object and a client part accesses the object by using the remote procedures provided by the server. An RPC-based program (in short, an RPC program) may consist of several servers and clients, and all these parts of the program work together concurrently on the programmer’s task. A server or client can fork to several processes if necessary.

2.2 Distributed IS Application Model

One important class of distributed applications is distributed information system applications. In a distributed IS application, there are usually a number of computers and processes managing some shared information, such as data bases. User programs access these computers and processes to obtain the information the user needs, or to update the stored information through these computers and processes. Time in a distributed IS application is not as strict a requirement as in distributed real-time applications. This paper is interested in distributed IS applications. From now on, when we mention distributed application, we mean distributed IS applications. Figure 1 is a generic model of distributed applications.

According to Figure 1, a distributed application consists of several client programs and several server programs. Usually a server program is located on a remote computer and a client program is located on the user’s (local) computer. A client program interfaces with the user, manages the local application...
process, and performs the communication between the client program and other related (remote) server programs (e.g., talking to the location brokers and servers). A server program usually manages an object (e.g., one part of a distributed database), performs the operations required by other programs, and manages the communications. Of course, the client program may also perform some operations directly on the local objects. This is not shown in the diagram because we want to emphasize the distributed characteristics of the application here. So, we can divide a distributed application into three parts:

- **User interface** deals with the interactions between the client program and the user.
- **Distributed frame** performs the communications among all the cooperative parts over the LAN.
- **Application modules** manage the objects and perform operations.

2.3 The Prototyping Model

Based on the distributed application model described in the last section, we indicate in Figure 2 a strategy for the development of distributed IS applications. According to this model, the prototyping process for a distributed application can be divided into three related activities:

- Distributed frame prototyping,
- user interface prototyping, and
- application modules prototyping.

One of the prerequisites of this computer-aided prototyping model is that all the related tools are available. When user requirements change, a new prototype should be ready for testing in a short time.

In this scheme, a distinct tool is provided for each prototyping activity. For example, when prototyping the distributed frame, a distributed frame generator is used to generate the program modules and to test drive programs; when prototyping the user interface, a user interface generator is used to generate the interface program from a straightforward definition file; and the application generator is used to generate the application modules. Now only data base management operations are provided in the application module generation. All these prototypes are relatively independent and can be tested separately with the participation of users. This gives the users a good chance to remove ambiguities in system requirements and to rethink and reorder their needs. After every part of the prototype is tested, their combination forms a working prototype of the whole program and can be tested again by users and devel-
opers. Further changes can be made easily because the computer-aided generators are available.

For each prototyping tool, the user is responsible of providing some definition files and, optionally, some user modules. The generators use definition files to generate all the other prototype programs, such as (server/client) driver programs, header files, stub files, and so on. The combination of the prototype programs and the user modules will make the prototypes executable. If the user does not provide user modules, the generators will produce some "dummy" user modules, so that the resulting prototypes can still be executable (see Sections 3.3 and 4.3, for more details).

Knowledge-based techniques (Mitchell et al., 1984) are used in implementing the distributed frame generator. It uses queries as well as server definition files to obtain information of the distributed frame from the user. The knowledge and transform mechanisms of the generator are implemented by using CLIPS (NASA, 1988; Giarratano, 1988), and the C language is used to help the implementation. The other two generators are implemented simply by using the yacc utility and the C language.

The general prototype generating process is indicated in Figure 3. At first the programmer creates one or more definition files according to the application. Then these files are sent to the appropriate generator where the test programs are generated. After that, the programmer can execute these test programs and make changes to the definition files on the basis of the results of the execution. Then, the changed definition files are sent to the generator again. When the programmer judges that the programs are correct, the final generation phase is chosen and the appropriate prototype modules are generated. By combining the distributed frame prototype, the user interface prototype, and the application module prototypes, the final distributed program is built up.

Three benefits can be obtained by using these prototyping tools. First, development time can be shortened. Second, design errors can be checked easily. Third, these tools provide more support for design changes during the development process.

Most existing prototyping systems only address part of the issues, such as menu generation, data base-oriented application generation, and software reusability, in developing distributed IS applications. TIEDUM (Blum, 1986) is a nice prototyping tool for developing interactive information systems. It maintains an application data base (ADB) that contains knowledge of the application. This includes a data model, program specifications, descriptive text, and help messages. From this ADB, it generates the executable system, prints the user documentation, and supports the maintenance process. But the system cannot be used directly in developing distributed applications. PSDL (Luqui and Kebabchi, 1988; Luqui and Lee, 1989) is a prototype-system description language. This language is integrated with a set of software tools, including an execution support system, a rewrite system, a syntax-directed editor with graphics capabilities, a software base, a design data base, and a design management system, to provide the capability of rapid prototyping. Reusability is one of the major advantages of the system. But there are no considerations on distributed systems. Two interface prototyping systems are mentioned in Section 4.1.

Prototyping distributed applications is the major concern in this paper. The combination of our three prototyping tools—the interface generator, the distributed frame generator, and the application modules generator—makes the rapid prototyping of distributed IS applications possible.

2.4 Distributed Calendar: The Example

We use a distributed calendar system as a nontrivial example. The system has three kinds of data bases (Zhou 1990a, b):

- A name data base: contains the names and corresponding UUIDs (Universal Unique Identifiers) of all legal users of the system.
- A meeting-room data base: If a meeting room is booked for a meeting, there will be an entry in this database that contains the room name, the meeting time period, and other information.
A set of calendar data base files: There is one calendar data base file for each group of users (e.g., all users within a department). If a user has a meeting, there will be an entry in his or her calendar data base file, which contains the user name, the meeting time interval, the participants, and other information.

For each data base, there is one server that maintains it. We call these a register server, a room server, and a calendar server, respectively. All these servers run "forever" in the network.

When a user logs into the system, the user name is used as a key to check his or her calendar data base location (UUID) from the name data base. The relevant calendar data base is then searched to see if there are any existing meetings for the user. When a user issues a meeting request, the system first checks the room data base to see if the room is available for the nominated period. If it is free, then it determines the calendar data base UUIDs of all the participants from the name data base and checks if they are all free during the meeting period. If the above check passes, an appropriate entry will be stored into each participant's calendar data base.

2.5 A Concurrency Primitive

In many cases it is desirable to have several RPC calls executed concurrently. For example, when we want to engage 10 participants in a meeting, we would like to have a facility that can make these 10 RPC calls at the same time instead of in sequence. We introduce a concurrency primitive called COBEGIN – COEND into the NCS. This primitive has the following format:

\[
\text{COBEGIN}(n);
\]
\[
\text{p}_1;
\]
\[
\text{p}_2;
\]
\[
\vdots
\]
\[
\text{p}_n;
\]
\[
\text{COEND}(n);
\]

where \( n \) is the number of concurrent RPCs in the primitive, and \( p_i, i = 1,2,\ldots,n \) are remote procedure calls. We can also express the remote procedure calls in the above primitive as follows:

\[
\text{COBEGIN}(n);
\]
\[
\text{for}(i = 1; i < = n; i + + )_{p_i};
\]
\[
\text{COEND}(n);
\]

The semantics of this primitive is that all \( n \) RPC calls are forked to different processes and executed simultaneously and are then joined in COEND. When all \( n \) RPC calls are completed (or have failed for some reason), the parent process continues.

When a user accesses several RPCs concurrently, he or she may want these RPCs to be executed as an atomic action. That is, either all of these RPCs are executed successfully or none of them is executed. The latter case corresponds to an error condition in one or more of the calls. The other successful RPCs in this case are rolled back. We discussed this in another paper (Zhou and Molinari, 1991). In this paper we do not consider the atomicy of these concurrent RPCs.

3. DISTRIBUTED FRAME GENERATOR

3.1 Syntax

The purpose of the distributed frame generator is to generate distributed frames for server and client programs according to server definition files (SDF). We give the syntax of a server definition file in Listing 1.

We use a modified BNF (Alagar, 1989) to denote the syntax of this and other definition files. In this notation, nonterminals are denoted in ordinary font and terminals are denoted in bold font, while the symbol ::= denotes defined as. Three operators are involved, namely:

- the construct \( \{ x \} \) means that \( x \) is replaced an arbitrary number of times,
- the construct \( \{ x \} \) means that \( x \) is optional, and
- the construct \( x[y|z] \) means that one of the items is selected.

\[
\text{SDF} ::= \begin{\text{BEGIN}}
\quad \begin{\text{HEADER}}
\quad \quad \begin{\text{ATTR}}
\quad \quad \quad \begin{\text{versionNo}}\text{integer}\end{\text{versionNo}}
\quad \quad \quad \begin{\text{CONSTS}}\text{CONST \{ CONST \}}\end{\text{CONSTS}}
\quad \quad \quad \begin{\text{CONST}}\text{variable = integer}\end{\text{CONST}}
\quad \quad \quad \begin{\text{STRUCTS}}\text{STRUCT \{ STRUCT \}}\end{\text{STRUCTS}}
\quad \quad \quad \begin{\text{STRUCT}}\text{structure-declarator}\end{\text{STRUCT}}
\quad \quad \quad \begin{\text{FUNCS}}\text{RPC Functions: RPCS RPC}\end{\text{FUNCS}}
\quad \quad \quad \begin{\text{RPC}}\text{Name: string ; PARAMS PARAM }\text{CLASS: declarator ;CLASS}\end{\text{RPC}}
\quad \quad \quad \begin{\text{PARAMS}}\text{in | out | in_out}\end{\text{PARAMS}}
\quad \quad \quad \begin{\text{PARAM}}\text{Param: CLASS: declarator ;CLASS}\end{\text{PARAM}}
\quad \quad \end{\text{ATTR}}
\quad \end{\text{HEADER}}
\end{\text{BEGIN}}
\end{\text{SDF}}
\]

Listing 1. Server definition file syntax.
The following notes apply to Listing 1.

(1) The nonterminals variable, integer, and string have the same meanings as in the C programming language.

(2) The nonterminal declarator has the same meaning as in the C programming language.

(3) The nonterminal structure-declarator is a simplification of the C struct definition, in that only simple types are allowed.

(4) Comments are allowed in the definition file. They are defined the same as the C programming language.

3.2 Semantics

A server definition file is defined as an optional HEADER part followed by FUNCTS part, with constants and data structure definitions also optional. The HEADER includes the server’s name, an optional interface attribute, a register string, a communication protocol, and a specification of maximum queue length. The server’s name is defined as a variable in the C language. This name will be used in the NIDL file as the interface name.

The interface attribute, if chosen, is a uuid followed by a version number. If the ATTR part is empty in the server definition file, the prototyping tool will generate an appropriate uuid and set the version number to 0, otherwise the user-provided uuid and version number are used. The version number is used by the NCS to distinguish between various versions of the same server. Only clients with the same version number as the server can access the server’s remote procedures, otherwise an exception will be raised in the client and no remote procedure will be executed in the server.

The register string is used by the server driver to register with the location broker. Usually it specifies what the server is going to do. When looking up the location broker, this string will be displayed together with other server information such as the interface uuid, host name, and port number. By looking at this string, one can determine the purpose of the server.

The communication protocol part is used to define which communication protocol is to be used. The NCS currently provides two kinds of protocols, namely, the DDS and IP (Kong et al., 1990; Zahn et al., 1990).

The maximum queue length part defines the maximum number of concurrent threads that can access the server. In NCS, the maximum concurrent accessing to a server is five. That is, at most five remote procedures can be accessed by one to five clients simultaneously. So, one to five can be defined here. If more clients than the defined maximum queue access the server, the clients which are later than the defined number will have exceptions raised.

The CONSTS part defined the constants used in the interface definition and is self-explanatory. The STRUCTS part defines the data structures used in the interface definition. It is almost the same as defined in the C language, except that only simple types, such as integer, character, string, and double, are allowed inside the structure. The FUNCTS part defines the remote procedures of the server. At least one remote procedure must be defined. Each remote procedure is defined as a name part and a parameter (PARAMS) part. The name of a remote procedure is simply a variable. There can be several parameters, each consisting of a class and a declaration. The class can be in, out, or in...out, which tells the NCS system that the parameter is used for input, output, or both, respectively. The declaration part is the same as in the C language.

For each remote procedure in a server definition file, we assign a sequential number (called the RPC number) to it according to the order in which it appears in the server definition file. The number starts from 0.

For implementation convenience, we put some restrictions in the syntax definition. First, some attributes of NCS are not included. For those attributes, we use typical default values. For example, we choose the registering of server to the global location broker as a necessary step in the server registration. This is actually what most real NCS based applications do. Also we predetermine the exception handling to simply display the exception message and exit to the upper calling level. If these are inappropriate it is easy to modify the text of the program prototype. Second, only simple type definitions and simple structure definitions are allowed. This may not fit some complex applications. But the programmer can still use this prototyping tool to test his ideas before starting the “real” programming, or use the prototypes produced by the prototyping tool as the draft design and do the expansion afterward.

The input to the generator is several server definition files, one for each server. We say the generator is in single-server mode if there is only one SDF input to the generator. If there are two or more SDFs input to the generator at the same time, then we say the generator is in multi-server mode. The output client driver can call any remote procedures of input servers sequentially or concurrently. Figure
4 indicates the input and output of the distributed frame generator (the *makefile* is not shown).

It is apparent that the multi-server mode includes all functions of the single-server mode. If there are several servers in a system under development, we usually first generate test programs for each server by using the single-server mode, and then test them one by one. If all servers are generated and tested, both sequentially and concurrently, we can group them together, use the multi-server mode to generate the test program for all these servers, and observe the execution of all these servers and their clients when running on different hosts. After all these tests have been passed, the final generation selection can be selected and prototypes for all servers and their clients can be generated for latter linking with other parts of the software.

3.3 Implementation

Without loss of generality, we describe implementation details for the single-server mode. After a programmer sends the server definition file to the generator, the generator first does syntax checking. If no errors are found, several program source files and a *makefile* are generated. The subsequent processing is specified by the *makefile*. That is, when using the *make* utility, at first four new files will be generated by NIDL compiler. They are the server stub, the client stub, the client switch (also used for communication), and a header file for both server and client drivers. After that the executable files of the server and client will be generated. Figure 5 indicates the structure of the processing.

The server driver does several jobs after the initialization. It first obtains a socket address from the operating system and registers itself with the location broker. It then does some jobs associated with exception handling. After that, it listens to its socket and responds to client calls. Some jobs are also performed before the server is shut down. We outline the action of server driver as follows:

1. Initialize.
2. Set communication protocol to that specified by the programmer, and obtain a socket from the system.
3. Register the server object with the location broker.
4. Set the exception handling subsystem.
5. Listen to the socket and respond to the client requests when necessary (sending the requests to the appropriate RPC procedures and obtaining the returns).
6. Perform necessary jobs before server shutdown, such as returning the socket to the system and unregistering from the location broker.

As we have indicated, in single-server mode the generated client driver can execute the server's remote procedures either sequentially or concurrently. If the server driver is running and the client driver is invoked, the client driver then first asks the user to input the number of concurrent remote procedures to be tested and their RPC numbers. The input parameters are these named remote procedures are then input from the keyboard. After that, these remote procedures are executed and results returned.

Although it is usually stated that a programmer can call a remote procedure in the same manner as a local procedure, the full calling process is actually much more complex than that of a local call. For each RPC call function, the client driver must do the following things after the input parameters are known:

1. Obtain the socket address of the server interface by interrogating the location broker.
2. Allocate (bind) an RPC handle according to the above address.
3. Allocate an exception handle and prepare the exception handling segment.
(4) Actually call the remote procedure (as for a local procedure call).
(5) Clean up the exception handler and display the returned information.

Of course if the server location is known the calling process will be correspondingly simpler. But usually a client program knows nothing about the location of a server that it wants to call. So, the above steps are necessary in general.

The termination of the server program also needs to be mentioned. After the server program is started, it will run forever unless the programmer kills its process or there exists a facility to terminate the server. Here we provide a facility to do that job. We add a remote shutdown procedure into the server and allow the remote shutdown of the server in the server program. Hence when the client driver calls the remote shutdown procedure of the server, the server will shut down itself and unregister from the location broker.

3.4 An Example

In our distributed calendar system, the first database designed is a calendar database. It is managed by a server called the calendar server. We use this as our example here. The following is the query process (a dialog with the distributed frame generator) for the server:

```
What is the program convenience consideration:
(best/good/moderate/low/do-not-care) best
What is the program speed consideration:
(highest/high/moderate/low/do-not-care) high
What is the program memory consideration:
(smallest/small/moderate/large/do-not-care) moderate
What is the program complexity consideration:
(smallest/small/moderate/large/do-not-care) moderate

How many different servers? 1
What is the No.1 server name? calendar
Where does the server locate:
(global/local/well-known/do-not-care) global
Which protocol is to be used:
(ip/dds/do-not-care) ip
Is the user provide server UUID (yes/no)? no
How many multi-accessing to the server allowed:
(1-5/do-not-care) 5
Is remote shutdown of server allowed (yes/no)? yes
Is server exception handling included (yes/no)? yes
Usage of RPC in client:
(sequential/parallel/do-not-care) parallel
Is client exception handling included (yes/no)? yes

Name of the server definition file: ca.def
```

The first four queries (in the first paragraph) are used for selecting different program modules and implementation methods that may affect the program's performance. The second paragraph is used to obtain some general information about the server(s). And the last paragraph indicates the input server definition file(s). The server definition file ca.def is given in Listing 2.

As we are using the single-server mode, Table 1 indicates the generated files.

The prototyping tool generates a UUID for the server and sets the version number to 0. The port number will be decided by the NCS dynamically when the server is registered with NCS. Also we added a remote procedure ca$shutdown at the end of the generated NIDL file. This will allow the client to shutdown the server remotely. The generated NIDL file ca.idl is indicated in Listing 3.

The generated server driver at first registers itself with the location broker, prints out the obtained socket information if the registration is successful,
Prototyping System for Distribution Information Systems

BEGIN

MAXNAMELEN = 20;
MAXVALLEN = 13;
MAXCMTLEN = 255;
MAXRESULTS = 20;

typedef struct {
    char username[MAXNAMELEN]; /* user name */
    char start[MAXVALLEN];    /* meeting start time */
    char end[MAXVALLEN];      /* meeting end time */
    char caller[MAXNAMELEN];  /* meeting caller */
    char comment[MAXCMTLEN];  /* memo of the meeting */
} DISCAL_REC;

RPC Functions:
Name: find_a_cal;
    /* find a meeting entry by name & start time */
Param: in: char name[MAXNAMELEN];
Param: in: char start[MAXVALLEN];
Param: out: DISCAL_REC rec; /* returned entry */
Name: find_all_cal;
    /* find all meeting entries for "name" */
Param: in: char name[MAXNAMELEN];
Param: out: DISCAL_REC rec[MAXRESULTS]; /* returned entries */
Param: out: int ret_num; /* returned number of entries */
Name: add_a_cal;
    /* add a meeting entry into the database */
Param: in: DISCAL_REC rec;
Name: del_a_cal;
    /* delete an entry from the database */
Param: in: char name[MAXNAMELEN];
Param: in: char start[MAXVALLEN];
Name: change_a_cal;
    /* change the contents of an entry */
Param: in: char name[MAXNAMELEN];
Param: in: char start[MAXVALLEN];
Param: in: DISCAL_REC rec; /* changed entry */

END

Listing 2. Server definition file for calendar server ca.def.

and then sets the error handling segment. After that, it listens to its socket port and serves its clients if necessary. That is, the server program will loop forever at the statement
rpc_listen(MaxCalls, &st).

If any client calls in, the server will serve them and then return to the loop. The maximum number
of concurrent calls to the server is defined in Max-
Calls, which in turn comes from the query process or server definition file. If the client call is the remote shutdown call, the server exits from the rpc_listen function and unregister itself from the location broker. After that, the server program exits. The generated server driver is in Listing 4:

The last statement of the server driver indicates the RPC template file or the complete remote procedure definition file (the user module). The generated RPC template file is very simple. For each remote procedure, a template is built. It specifies the procedure's name, parameters and their types, and a procedure body that prints out a sentence stating the name of the procedure. We show one of these procedure templates as follows. Others are similar.

Table 1. Output Files for ca.def

<table>
<thead>
<tr>
<th>File name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca.idl</td>
<td>NIDL file of the server</td>
</tr>
<tr>
<td>ca_ser.c</td>
<td>Server drive program</td>
</tr>
<tr>
<td>ca_rpfuns.c</td>
<td>RPC template file</td>
</tr>
<tr>
<td>dp.c</td>
<td>Client drive program</td>
</tr>
<tr>
<td>ca_includes.h</td>
<td>Header file for server and its client</td>
</tr>
<tr>
<td>make_dp</td>
<td>Make file for testing programs</td>
</tr>
</tbody>
</table>
The generated client driver enters a dialog with the user. It first requests the number of RPCs to be tested in the current execution. It then asks the input of the RPC numbers for each server, as well as the input arguments of each RPC call. After verification of this input information and the setting of exception handling segment, the drive calls an RPC parallel executive function parRun. If there is only one RPC call to be executed, the single RPC call execution function is called by the parRun function. Otherwise, the COBEGIN-COEND primitive is used to execute the RPC calls. Listing 5 is the client driver.
/* Server program file of server "ca" */
#include "ca_include.h"

Several utility functions are omitted here.

static lb_entry_t lentry;

int SocketAddrRegister()
{
    status_st st;
    socket_saddr_t loc;
    char name[256];
    unsigned long namelen = sizeof(name);
    unsigned long port;
    extern rpc_senv_t ca$server_epv;
    /* Some NCS function calls that allow remote shutdown server,
     set protocol, and obtain socket and port (assigned to
     "name" and "port") are omitted here.
     */
    printf("Got socket: %s[%lu]%n", name, port);
    /* register to the location broker */
    rpc_Register(&ca$if_spec, ca$server_epv, &st);
    if (st.all != 0)
    {
        printf("Cannot register to RPC library - %s%n", error_text(st));
        return -1;
    }
    lb_Register(&uuid_snif, &uuid_snil, &ca$if_spec_id, OL,
                &RegName, &Loc_sa, Loc_len, &lentry, &st);
    if (st.all != 0)
    {
        printf("Cannot register to GLB - %s%n", error_text(st));
        return 0;
    }
}

int UnregisterBroker()
{
    status_st st;
    lb_Unregister(&lentry, &st);
    if (st.all != 0)
    {
        printf("Cannot unregister from GLB - %s%n", error_text(st));
        return -1;
    }
    rpc_Unregister(&ca$if_spec, &st);
    if (st.all != 0)
    {
        printf("Cannot unregister from RPC library - %s%n", error_text(st));
        return -1;
    }
    return 0;
}

main(argc, argv)
{
    char *argv[];

    status_st st;
    pfm_Select_opt_rep rep;
    /* socket preparation - get socket, register ... */
    if (SocketAddrRegister() != 0)
    {
        printf("Error in socket preparation.%n");
        exit(1);
    }

Listing 4. Server driver file for calendar server ca.def.
The header file for both server and client drivers contains the necessary header files for general processing and RPC-based processing. It also contains some of the definitions obtained from the query or server definition file. The header file is indicated in Listing 6.

When using the make utility to construct the executable files, the NIDL compiler will generate the files shown in Table 2.

Finally two executable files will be created. They are: CA_Ser.R (linked with ca_ser.o and ca_sstub.o), which is the server-executable file, and dp_x (linked with dp.o, ca_cstub.o, and ca_cswtch.o), which is the client-executable file. Because there are five remote procedures (not counting the shutdown procedure here), it is possible to execute them sequentially and concurrently. That is, the user can test this through the client driver.

By using the generator, it is very easy to cope with RPC interface changes. For example, consider the interface for remote procedure add_a_cal to be changed to

```c
ca$add_a_cal(  
  handle_t [in] h,  
  DISCAL_REC [in] rec  
  int [out] existing_num;  
);
```

In the new RPC template the new segment will be

```c
/* Rpc "add_a_cal" call function of "ca" */  
ca$add_a_cal(h, rec, existing_num)  
handle_t h;  
DISCAL_REC rec;  
int *existing_num;  
{
  printf("This is RPC function ca$add_a_cal.\n");
  
  /* The contents of the RPC function */
}
```

Finally in the client driver, one of the changed segments is the remote procedure calling function.

If there were no generator, the programmer would have to change the NIDL file, the server's remote procedure definition file, and the client file. With the help of the generator, the user only changes the server definition file. The generator will then produce all the files for the modified situation. Of course, if the real work is to be done, the programmer has to change the user module as well.

By the help of the generator, the programmer can build the prototype programs of the distributed frame for the proposed distributed application in a
Prototype System for Distribution Information Systems

/* Client program file of parallel testing */
#include "ca_includef.h"
#include "mhead.h"
define NumOfServers 1
int NumOfEach[NumOfServers] = {
1,
};
define NumOfRpcs 6
char *ServerNames[NumOfServers] = {
"ca",
};
typedef struct parpc
  int server; /* server number */
  int rpcnum; /* RPC number */
  int argc; /* number of input arguments */
  void *argv[20]; /* argument array, maximum is 20 */
} PARRPC;
PARRPC *rpcin[MAXRPCS];

char * *RpcNames[NumOfServers] = {
  "find_a_cal", "find_all_cal", "add_a_cal", "del_a_cal", "change_a_cal", "Shutdown",
0,
};
char **RpcNames[NumOfServers] = {
  RpcNames0,
};

/* Some utility definitions and functions are omitted here. */
/* get server location by interface, take the 1st one */
static handle_t GetServerLoc(serNo)
int serNo; /* server number */
{   
  lb_Slookup_handle_t ehandle = lb_Sdefault_lookup_handle;
  static lb_Sentry_t locs[5]; /* maximum locations: 5 */
  static unsigned long n_locs; /* # of locations */
  static int loc_i = 0; /* current index */
  status_st st;
  handle_t rh;
  loc_i = 0;
  lb_Slookup_interf(&SerInterf(serNo), &ehandle, 5, &n_locs, locs, &st);
  if (st.all ! = status_Sok) {
    fprintf(stderr, "Cannot locate server - %s\n", error_text(st));
    return (handle_t) 0;
  }

  /* several calls which find the first valid location in locs[] and bind it to rh are omitted here. */
  return rh;
}

/* Rpc "find_a_cal" call function of "ca" */
int ca_find_a_cal_func(rpcnum, argc, argv)
int rpcnum;
int argc;
void *argv[];
{   
  handle_t rh;
  status_st st;
Listing 5. Client driver file for calendar server ca.def.

73
pfm_scleanup_rec rec;
char name[MAXNAMELEN];
char start[MAXVALLEN];
static DISCAL_REC rec;
/* get the server location */
/* for saving time, the ServerHandle[0] can be used */
rh = GetServerLoc(0);
if (rh == NULL) return -1; /* cannot get location */
sst = pfm_scleanup(rec);
if (sst == NULL) {
    return -1;
}
/* converting "in" parameters */
conv_str(name, argv[0]);
conv_str(start, argv[1]);
/* actually call the remote procedure */
csa_find_a_cal(rh, name, start, rec);
print("find_a_cal returned\n");
/*
return parameter checking here (omitted)
*/
pfm_srls_cleanup(rec, sst);
return 0;
}

/* Other remote procedure call frames are similar to the above, and are omitted here. */
/* "ca": Rpc "find_a_cal" call use function */
static void ca_find_a_cal_use(rpcnum, args, argv)
int rpcnum;
int args;
void *argv[];
{ 
    if (ca_find_a_cal_use(rpcnum, args, argv) == -1) {
        printf(stderr, "Cannot communicate with server -ca\n");
    }
}
/* Other remote procedure call frames are similar to the above, and are omitted here. */
void (*RrpcUseFuncs0 [])() = {
    ca_find_a_cal_use,
    ca_find_all_cal_use,
    ca_add_a_cal_use,
    ca_del_a_cal_use,
    ca_change_a_cal_use,
    ca_shutdown_use,
    0
}

Listing 5—Continued.
Some parallel execution-related functions are omitted here.

/*
   * parallel execution rpcs */
int num;  /* number of RPCs */
{
    int sernum, rcnum, argc, i;
    if (num <= 0)
        return;
    if (num == 1) {
        sernum = rpcin[0]->sernum;
        rcnum = rpcin[0]->rcccnum;
        argc = rpcin[0]->argc;
        (**RpcUseFuns[sernum][rcnum])(rcnum, argc, arg);
        return;
    }
    else {  /* I<num<=NumOfRpcs */
        /* implemented calling format of a parallel RPC call */
        cobegin(num);
        for (i=0; i<num; i++)
            execut2(rpcin[i]->sernum, rpcin[i]->rcnum,
                     rpcin[i]->argc, rpcin[i]->arg);
        coend(num);
    }
}

Some input-related functions are omitted here.

/*
   * main function */
main(argc, argv)
int argc;
char *argv[];
{
    int prpc, i, j, k;
    status_set fut;
    pfm_scleanup_rec rec;
    allocPar(); /* address allocation */
    if (argc < 2) usage(-1);
    prpc = atoi(argv[1]);  /* Number of parallel RPCs */
    for (i=0; i<prpc; i++)
     /* obtain the ith RPC info into rpcin[i] structure */
        if (getDetail(i) == -1) exit(0);

    /* Some functions that verify the testing message are omitted here. */
    /* set exception segment */
    pst = pfm_scleanup(rec);
    if ((fst != NULL) && (pfm_scleanup_set) {  
        if ((fst != status_set)
            fprintf(stderr, "Exception raised - %d\n", error_text(fst));
            pfm_signal(fst);
}
    parRun(prpc);
    pfm_Sexit();
}
Listing 5—Continued.
 very short time. Of course the distributed frame is of no use if real jobs cannot be performed in the RPC template file (in that case, the testing is also of no sense) or if a suitable user interface cannot be employed. The next two sections will address some aspects of the application modules and user interface prototyping.

### Table 2. Files Generated by NIDL Compiler

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca.h</td>
<td>Header file generated by NIDL</td>
</tr>
<tr>
<td>ca_cstub.c</td>
<td>Client stub file generated by NIDL</td>
</tr>
<tr>
<td>ca_cswich.c</td>
<td>Client switch file generated by NIDL</td>
</tr>
<tr>
<td>ca.stub.c</td>
<td>Server stub file generated by NIDL</td>
</tr>
</tbody>
</table>

4. USER INTERFACE GENERATOR

4.1 Introduction

User interface prototyping has been discussed by several researchers. Christensen and Kreplin (1984) described a user interface prototyping system that can generate prototypes from specification files in a dialog format. In a specification file, the *dialog structure* is defined. Arthur (1987) described a formal menu-based system. A *hierarchical structure* is defined in his menu specification.

Intuitively, a menu system provides users with a set of selections and for each selection made executes an associated action. The selected action can be a function or can be another menu. So the
Prototyping System for Distribution Information Systems

execution cycle is:

1. Initialize the main (first) menu to the current menu.
2. Display the current menu.
3. Obtain the user response.
4. If the response is not valid, go to (2).
5. If the selected selection is a function, execute it and then go to (1).
6. If the selected selection is a child menu, set it to the current menu and go to (1).

The situation can be modeled as follows. Let \( C = \{ c_1, c_2, \ldots, c_n \} \) be the set of all menus defined in the interface file. Let \( P = \{ p_1, p_2, \ldots, p_m \} \) be the set of all functions defined in the interface file. We also add two more functions into \( P \): a null function (it does nothing) and an exit to parent function. Now we define that if \( p_i \in P \), then there is a \( c_j : C \) such that \( p_i \) is one of the selections of \( c_j \). Let \( A = \{ a_1, a_2, \ldots, a_k \} \) be the set of user responses to the interface. The interface system can be viewed as a mapping:

\[ r : C \times A \rightarrow C \times P. \]

By \( r(c_i, a_j) = (c_k, p_i) \) we mean that when the menu \( c_i \) is displayed and the user selects action \( a_j \), the system responds with a (probably) new menu \( c_k \) and an action \( p_i \). If the selected menu item is a function, then we have \( i = k \) and \( p_i \) is a real function, which will be executed after the selection. If the selected item is another child menu, then \( j \neq k \) and \( p_i \) is a null function. If the \( c_i \) is the main menu and termination is selected, the \( p_i \) function then provides an exit from the system.

Let \( r(c_i, a_j) = (c_k, p_i) \). If \( a_j \) is not the exit to the parent menu function, then we say that \( c_i \) is a parent menu of \( c_k \) and \( c_k \) is a child menu of \( c_i \). A history of a menu \( c \) is its parent path from the main menu (the root) to \( c \). Sometimes the user may wish to know the history of a current menu, especially when the menu path is too deep.

4.2 Syntax

The definition of the interface definition file syntax is as follows:

The nonterminals variable, integer, and string have usual meanings as in the C programming language.

4.3 Semantics

We call the integer following the menu in MENU-DEF (Listing 7) the menu number. The following notes indicate the semantic issues:

1. All menu numbers must be distinct.
2. Any child menu can have only one parent menu.
   That is, if a menu number is used as a child menu number within a menu selection, it cannot be used in any other menu selection.
3. When the menu is used to generate command level interface instead of window-oriented inter-

### Listing 7. Interface definition file syntax

```
MENU ::= menu variable

MENU_DECLS := MENU_DEF [ ; MENU_DEF ]

MENU_DEF ::= menu integer

HEAD_DEF ::= DIR ; POS ; [ EXE ] ; [ COLOUR ]

DIR ::= direction : D_PARAM

D_PARAM ::= horizontal | vertical

POS ::= position : P_PARAM

P_PARAM ::= default

EXE ::= execution : FUNC

FUNC ::= variable() ;

COLOUR ::= foreground = integer , background = integer

SEL_DEF ::= selections ;

ONE_SEL ::= [ ONE_SEL ]

SEL_ITEM ::= SEL ITEM , HELP ITEM

HELP_ITEM ::= string , CLASS

CLASS ::= child menu integer | function FUNC

HELP_ITEM ::= help string
```

77
face, the position definition, direction definition, and color definition are not applicable.

(4) The EXE part definition provides a function name that may be called when the menu is selected. After such a call, control returns to the menu driver. This gives the user a lot of flexibility when he wants to do something together with the menu display.

(5) If the window position is defined to be smaller than the length of all menu selections of the menu, then only part of the menu items are displayed. The others can be viewed by using arrow keys. When vertical is used in the definition of a menu, then the lower row and column definitions are not applicable. The system will adjust them according to the selection items.

The user interface generator is used to generate the window-oriented or command-oriented interface screen to accept the user commands. It reads an interface definition file and translates the file into three related files. The arc: a header file, which contains the definitions of the user interface, an interface drive program (called the menu driver), which can be used to drive the execution of the interface program, and a dummy interface function module (user module), which defines all the interface functions used in the menu drive in dummy format. The definition files are self-explanatory and the generated files are ready to execute after compiling. Figure 6 indicates the input and output files of the user interface generator.

typedef struct node /* for holding menu info */
    int num; /* node number */
    int numu; /* count of sons */
    int numb; /* brother number */
    char *name; /* menu name */
    char *help; /* help information */
    int class; /* node class -- leaffunction */
    int ndir; /* menu direction */
    int bord; /* border type */
    int urow; /* row position of upper-left */
    int ucol; /* col position of upper-left */
    int lrow; /* row position of lower-right */
    int lcol; /* col position of lower-right */
    int fcolor; /* foreground colour */
    int bcolor; /* background colour */
    char *fname; /* function name -- leaf node */
    char **snslst; /* son menu name list */
    char **hlplist; /* son help information */
struct node *pare; /* parent menu */
struct node *son; /* first son menu */
struct node *prev; /* previous menu */
struct node *next; /* next menu */
} TNODE;

4.4 Implementation

After the processing, the menu header file will contain all the static menu definitions obtained from the interface definition file. The menu data structure is called TNODE and is defined as in Listing 8.

So, a menu structure is defined as a multi-linked tree. Each node has several attributes. The parent menu points to the menu that contains the current menu as a child menu; first son menu points to the first selection item; previous menu points to the left brother; and next menu points to the right brother of the current menu.

Figure 7 illustrates the meaning of these pointers. For menu node n, its pare pointer points to menu node a because n is one of a’s sons. Its prev pointer points to menu node b because b is its left brother. n’s fson pointer points to the its first sons menu, d, and its next pointer points to its right brother c.
the interface function module is similar. Following is
the dummy module for function

```
create();
/* function module */
#include <stdio.h>
/* function create() */
create()
{
    printf("This is function create(). \n");
}
/*
other function definitions are omitted
*/
```

If the "WINDOW" option is set in the command
when generating the interface, then the interface
program will first display the menu as Figure 8a. If
the user selected MEETING, the menu screen will
change to Figure 8b. When Create is selected, the
associated function create() will be executed.

If the "COMMAND" option is set in the command
when generating the interface, then the interface

```
/* user interface definition file for calendar */
/* (for testing) */
menu window-oriented
menu 0
    direction:  horizontal;
    position:  default;
    selections:
        "MEETING",  child
        help "Enter a new meeting.";
        "REMINDER",  child
        help "Enter a new reminder.";
        "SHOW",  child
        help "Show my engagement."
end selections
end menu 0;
menu 1
    direction:  vertical;
    position:  default;
    selections:
        "Create",  function create()
        help "Input new meetings";
        "Edit",  child
        help "Modify existing meetings"
end selections
end menu 1;
/*
definitions for menu 2, 3, ... etc. are defined here
*/
```

Listing 9. Interface definition file example.
#include "keys.h"
#include "menuprog.h"
#include "cal.h"
#include "stdlib.h"
#include "s_menu.h"
#include "msckey.h"

/* find the ith son of node np */
TNODE *find_son_i(npi, i)
TNODE *np;
int i;
{
    if (np != 0)
    {
        np = np->son;
        while (np != 0 && np->bnum != i)
            np = np->next;
    }
    return np;
}

menu_0(npi)
TNODE *np;
{
    int i, cn, first_time;
    TNODE *snp;
    cn = 0;
    first_time = YES;
    while (1)
    {
        i = menu_sub(np, first_time, cn);
        switch (i) {
        case 0:
            snp = find_son_i(np, i);
            cn = menu_1(snp);
            break;
        case 1:
            snp = find_son_i(np, i);
            cn = menu_2(snp);
            break;
        case 2:
            snp = find_son_i(np, i);
            cn = menu_3(snp);
            break;
        case EXIT:
            return (np->bnum + ESCAPE);
        default:
            break;
        }
    }
}

menu_1(npi)
TNODE *np;
{
    int i, cn, first_time;
    TNODE *snp;
    cn = 0;
    first_time = YES;
    while (1)
    {
        i = menu_sub(np, first_time, cn);
        switch (i) {
        case 0:
            create();
            break;
        case 1:
            snp = find_son_i(np, i);
            cn = menu_4(snp);
            break;
        case LEFT:
            return upper_choice(np, i);
        case EXIT:
            return (np->bnum + ESCAPE);
        default:
            break;
        }
    }
}


case 1:
    snp = find_son_i(np, i);
    cn = menu_2(snp);
    break;

case 2:
    snp = find_son_i(np, i);
    cn = menu_3(snp);
    break;

case EXIT:
    return (np->bnum + ESCAPE);
default:
    break;

other program segments are omitted here
*/
main(argc, argv)
int argc;
char **argv;
{
    menu_0(menu_tree_node);
}
char *menu_0_name_list[] = {
   "MEETING",
   "REMINDER",
   "SHOW",
   0
};
char *menu_0_help_list[] = {
   "Enter a new meeting.",
   "Enter a new reminder.",
   "Show my engagement.",
   0
};
char *menu_1_name_list[] = {
   "Create",
   "Edit",
   0
};
char *menu_1_help_list[] = {
   "Input new meetings",
   "Modify existing meetings",
   0
};

/*
other menu names and help lists are omitted here
*/

TNODE menu_tree_node[] = {
   { 0, 0, 3, 0, "MAIN", 0, 1, 0, 1, 0, 2, 79, 0,
      7, 0, menu_0_name_list, menu_0_help_list, 0,
      &menu_tree_node[1], 0, 0 },
   { 1, 1, 2, 0, "MEETING", 0, 1, 1, 1, 2, 4, 0, 0,
      7, 0, menu_1_name_list, menu_1_help_list,
      &menu_tree_node[0], &menu_tree_node[2],
      0, &menu_tree_node[7] },
   { 2, -1, 0, 0, "Create", 0, 0, 0, 0, 0, 0, 0,
      0, 7, "create()", 0, 0,
      &menu_tree_node[1], 0, 0, &menu_tree_node[3] },
   { 3, 4, 3, 1, "Edit", 0, 1, 1, 1, 2, 4, 0, 0,
      7, 0, menu_4_name_list, menu_4_help_list,
      &menu_tree_node[1], &menu_tree_node[4],
      &menu_tree_node[2], 0 },
   { 7, 2, 1, "REMINDER", 0, 1, 1, 1, 2, 18, 0, 0,
      7, 0, menu_2_name_list, menu_2_help_list,
      &menu_tree_node[0], &menu_tree_node[8],
      &menu_tree_node[11], &menu_tree_node[10] },
   { 10, 3, 2, 2, "SHOW", 0, 1, 1, 1, 2, 30, 0, 0,
      7, 0, menu_3_name_list, menu_3_help_list,
      &menu_tree_node[0], &menu_tree_node[11],
      &menu_tree_node[7], 0 },

/*
other menu tree node information are omitted here
*/
};
program will display at first the menu as:

```
Menu MAIN has the following selections:
  0. EXIT   ( Exit to OS or the upper level menu.)
  1. MEETING ( Enter a new meeting.)
  2. REMINDER ( Enter a new reminder.)
  3. SHOW   ( Show my engagement.)
Your choice:
```

If you choose 1, then the following menu will be displayed:

```
Menu MEETING has the following selections:
  0. EXIT   ( Exit to OS or the upper level menu.)
  1. Create ( Input new meetings)
  2. Edit   ( Modify existing meetings)
Your choice:
```

Again, if you choose 1 at the MEETING menu level, then the function create() will be executed. Of course, if only the dummy functions are linked with the menu driver, then no real work can be done except a message is printed to indicate that the create() function is executed.

If there is any change in the interface definition, the developer can simply change the definition file and regenerate the source files.

5. APPLICATION MODULE GENERATOR

5.1 Introduction

Currently we only implement a database application generator that can produce simple database-oriented programs. The main purpose here is to generate RPC functions for servers that perform database-oriented operations and to generate some screen layout modules for client programs. The algorithm implementation for client and server programs must rely upon the developers at this moment.

Database management is one of the important issues in distributed IS application. A lot of database systems available today provide convenient interfaces to upper level programs. We selected the

Apollo's Database Access Manager (Perry, 1989) as our underlying database system. It has very good performance in lightweight database applications and has been used in many distributed applications.

5.2 Syntax

A database description file defines the fields of each database file, the screen layout of each database file, and operations to be performed on these database files. The application generator generates the database files and program modules for database operations and screen layouts. The formal description of the database definition file is indicated in Listing 12.

5.3 Semantics

A database description file consists of a file definition part and an operation definition part. The optional CONSTS and STRUCTS parts are the same as in server definition file. One or many database files can be defined in a single description file (this may make the joint operations possible). Each database file definition lists all its field names, types of fields, lengths of the fields, prompts and templates used, and printing positions for the prompts and templates. If key is specified within a field, then the field is to be used as an index field for the database file. Table 3 defines some field types. Note that a field can be in right or left justified format by specifying its JUSTIFYING definition (it is unjustified by default).

The operation definition part defines all the basic operations over the defined database files. At this moment we have only implemented several simple operations over a single database file, such as add an entry, change an entry, delete an entry. Complex

---

Figure 8. The menu screens.
DATABASE ::= $\texttt{define variable} \\
[ \texttt{CONSTS} ] \\
[ \texttt{STRUCTS} ] \\
\texttt{FILE} ; ( \texttt{FILE} ; ) \\
\texttt{OPERATIONS} \\
\texttt{define variable} \\
\texttt{FILE} ::= $\texttt{file variable} \\
\texttt{FIELD} ( ; \texttt{FIELD} ) \\
\texttt{file variable} \\
\texttt{FIELD} ::= $\texttt{FD_NAME, TYPE, LENGTH, PROMPT, TEMPLATE, PRMPT_ROW, PRMPT_COL, FIELD_ROW, FIELD_COL} \\
[ , \texttt{key} ] \\
\texttt{FD_NAME} ::= \texttt{variable} \\
\texttt{TYPE} ::= $\texttt{TP [ JUSTIFYING]} \\
\texttt{TP} ::= A | C | D | N \\
\texttt{JUSTIFYING} ::= R | L \\
\texttt{LENGTH} ::= \texttt{integer} \\
\texttt{PROMPT} ::= \texttt{string} \\
\texttt{TEMPLATE} ::= \texttt{string} \\
\texttt{PRMPT_ROW} ::= \texttt{integer} \\
\texttt{PRMPT_COL} ::= \texttt{integer} \\
\texttt{FIELD_ROW} ::= \texttt{integer} \\
\texttt{FIELD_COL} ::= \texttt{integer} \\
\texttt{OPERATIONS} ::= \texttt{operations} \\
\texttt{OPERATION} ( ; \texttt{OPERATION} ) \\
\texttt{operations} \\
\texttt{OPERATION} ::= \texttt{OP_NAME, OP, FILE_NAME} \\
\texttt{OP_NAME} ::= \texttt{name: variable} \\
\texttt{OP} ::= ADD | CHG | DEL | FDA | FDG \\
\texttt{FILE_NAME} ::= \texttt{variable}

Listing 12. Data base definition file syntax.

operations such as joint operations are designed but not implemented.

The input and output files of the application module generator can be indicated as in Figure 9. From
the definition file, the generator produces an application driver program, a data base operation module,
screen layout operation model, and some header files.

5.4 Implementation

The following listing provides the data base definition file for our calendar data base system. Please notice
that the constant definition and data structure definition are the same as in server definition file case. So they are actually in a header file and are included by both definition files. The syntax in Listing 12 applies to the file after this inclusion has been
performed.

The pseudocode of the main loop of the application driver is as follows:

```
main_loop()
{
    while (TRUE)
    {
        list all operations for selecting;
        if (selected a valid operation)
            do it;
        else
            if (selected EXIT)
                break;
    }
}
```

The user interface generator is used by the application modules generator to produce the listing and

Table 3. Field Type Definition

<table>
<thead>
<tr>
<th>Field type</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>Alphabetical</td>
</tr>
<tr>
<td>C</td>
<td>Currency</td>
</tr>
<tr>
<td>D</td>
<td>Date</td>
</tr>
<tr>
<td>N</td>
<td>Numerical</td>
</tr>
</tbody>
</table>

Figure 9. Input and output files of the application module generator.
selections for the operations defined in the data base definition file. The fields of each data base are displayed on the screen according to the definition file. Because our emphasis here is the distributed frame generation, this part of the model is currently incomplete.

5.5 Interfacing the Generators

One may think that dividing a distributed application into the distributed frame part, the user interface part, and the application modules part and generating them independently is unnecessary—some extra efforts are needed for connecting all these independently generated prototypes. For small programs that may be true. But as we have mentioned at the beginning of this paper, we are interested in programming-in-the-large, because a distributed application is usually a large program. It needs a number of people to work together. In that case, dividing a complex design into several smaller designs is preferable. So we need some connection facilities to connect these prototypes.

As indicated in Figure 10, three connections are needed. In the user interface and distributed frame connection, the user interface will use the remote operations provided by the distributed frame, but the distributed frame usually does not use functions from the user interface. So a header file is used to provide the user interface with all available remote procedure calls exported by all related servers from the client viewpoint.

In the distributed frame and application module connection, the server programs need to use the application modules to operate on objects. But an application module usually does not use functions from the distributed frame. So, a header file is used to provide the distributed frame with all the functions in the application module.

In the user interface and application module connection, the user interface may use the functions of the application modules to perform some local operation on local objects. So, a header file is also provided for that purpose.

6. PROTOTYPING THE DISTRIBUTED CALENDAR SYSTEM

Next we describe the process we used to develop the distributed calendar system with the help of our model. The system was designed to be used in a LAN, which consisted of hundreds of Apollo workstations and it is assumed that all of them have NCS installed. At the beginning, it was intended to man-

Figure 10. Generator connections.

age the following functions:

1. Given the data of a meeting, we want the meeting to be arranged in each related person’s calendar if all participants of the meeting are free during the meeting period.
2. Allow a person to set some personal reminders into his or her own calendar.

At first we identified the following concepts:

• Caller: A person that issues the meeting.
• Callee: One of the persons that a caller wants to have a meeting with.
• Meeting data: A message that includes the caller’s name, the callee’s name, meeting time interval, and a brief description of the meeting purpose.
• Calendar data base: If a user has a meeting, there will be an entry in his or her data base that contains the meeting data.
• The process of reminders is the same as for calendars except that the meanings of the fields are changed to the user’s name, the reminder’s time, and reminder message.

Based on these assumptions and requirements, the first prototype was developed and tested by using our prototyping tools. It consisted of the following parts:

• A data based management module that manages the accessing of the calendar data base located in the local host.
• A calendar server that manages the calendar data base for a group of users through the data base management module and exports the data base operations to remote clients.
• A client program that can access the calendar data base through the operations of the calendar server; the client program can be executed in several hosts simultaneously.
A user interface that helps users to perform the desired operations over the calendar database.

The creation of the first prototype was not difficult with the help of our prototyping tools. The only place that needed more attention was the client program's algorithm, because the prototyping system did not support this function.

A research group manager and a sales person acted as users (the user). After executing the prototype together with the user, it is identified that there should be a meeting room database because room conflicts are also a problem. That is, two meetings cannot be assigned to the same meeting room with intersecting time intervals. So a room database was added, and a new field meeting_room was added into the calendar server definition file. That caused the distributed frame prototype to be regenerated to have the meeting room server working cooperatively with the calendar server. Also the new client can now use both data bases simultaneously. Several new modules were added into the user interface prototype to fit the new data base; and some new application modules were added. Also, the first requirement was changed to:

(1) Given the data of a meeting, we want the meeting to be stored in the meeting room data base and to be arranged in each related person's calendar if the meeting room and all participants of the meeting are free during the meeting period.

From the first prototype to the second prototype, most changes were performed by rewriting the various definition files and only a small amount of code rewriting was needed.
Next, the user discovered that after a meeting is set up, there is no way to cancel it. But in real life such a requirement obviously arises. So, a function was added into the application prototype that a caller can cancel a meeting he has entered. That is, the third requirement was added:

(3) Allow a caller to delete a meeting (which he or she created) from the meeting room data base and from all participants' calendars.

After further reprototyping and testing, the user and the developer realized that there should be a function to delete all old meeting data, that is, the meetings that had been held. That function then was added to the data base manager. It checks the data base periodically to delete all old entries. That is, the fourth requirement was added:

(4) Automatically delete all old meeting entries (that is, the meeting ending time is earlier than the current time).

When the system was put in use by some users, they found that there should be some restrictions on the issuing of meetings. That is, some priorities are needed among all users. So, the system was changed again that each user was assigned a priority number. In that case, only the users with higher or equal priorities can arrange meetings with users having lower or equal priorities. Because there had been a user information server existing in the LAN, an interface with that server was built and two fields were added into the data base managed by the user information server. These two new fields specify the user priority and the user calendar server location, respectively. At last, the system was ready for use.

Not surprisingly the computer-aided rapid prototyping method will reduce the distributed IS application development cost. For example, in a reported prototyping experiment (Boehm et al., 1984), systems were developed at 40% less cost and 45% less effort than conventional methods. Other researchers have reported even more impressive figures (Hekmatpour and Ince, 1988). As in our example, only 1 man-week was used for the initial prototype while conventional methods were estimated to have required 4 man-weeks.

7. REMARKS

A set of rapid prototyping tools that can effectively help the development of distributed programs are described in this paper. Three benefits can be obtained from these prototyping tools. First, it is much quicker and easier for a programmer to write a definition file for a program generator than to write the program itself. Second, it is easy to make mistakes during programming. With definition files, however, the possibility of making mistakes is smaller than with general programming. This is due to the fact that a definition file is simpler than the equivalent program, and there is some syntactic and even semantic checking during prototype generation. Third, the prototyping tools provide more support for changes during program development. As is well known, accommodating changes to the system requirement and design is an intrinsic property of software development.

The following expansion works are underway. We are expanding the tools to SUN/RPC and other RPC environment. One graduate student is combining the three definition languages into one definition language. Another graduate student is developing a lightweight concurrent system to support a group of concurrent RPC calls and to make the concurrent RPC calls as atomic actions. A window-based interface generator and a sophisticated application model generator that can generate INGRES data base application models are also under development.

ACKNOWLEDGMENT

I thank Dr. Brian Molinari for his valuable advising and many discussions.

REFERENCES


86
Prototyping System for Distribution Information Systems


Biographies

Jen-Yen Chen received a B.S. degree in industrial engineering from Tung-Hai University, Taichung, Taiwan, an M.S. in industrial engineering, an M.S. in computer science, and a Ph.D. in computer science from the University of Texas at Arlington. He is currently an associate professor in the Department of Computer Science and Information Engineering at Chiao-Tung University, Hsinchu, Taiwan. His research interests include software process modeling, object technology, information systems, and software engineering environments.

Anna Hata is an associate professor in the Electrical Engineering Department at the University of Hawaii at Manoa, Honolulu, Hawaii. She received her M.S. and Ph.D. in computer science from the Department of Electronics at the Warsaw University of Technology, Poland, in 1977 and 1982, respectively. She was a visiting scientist at the Imperial College, University of London, England, a postdoctoral fellow at the University of California at Berkeley, an assistant professor of computer science at the Johns Hopkins University, and a technical staff member at AT&T Bell Laboratories. Her research contributions include system and workload modeling, performance analysis, reliability, modeling process synchronization mechanisms for distributed systems, distributed file systems, distributed algorithms, congestion control in high-speed networks, and reliable software architecture for switching systems. She is a senior member of the IEEE and a member of the ACM.

Yu-Shiang Hung received a B.S. and an M.S. in computer science from Chiao-Tung University, Hsinchu, Taiwan. He is currently working for his Ph.D. degree in computer science. His research interests include object-oriented analysis, design methods, and information modeling.

Paul Quintas is a senior research fellow at the Science Policy Research Unit (SPRU), University of Sussex, United Kingdom. With a long-standing interest in the social, economic, and political processes surrounding technological change, he has researched and published in the areas of the software development process; technology and R&D policy; firm-level technology strategies and the management of innovation; interfirm R&D collaboration; industry-academic links; the social and economic implications of information technology, science parks and regional technology and industrial policy. He has acted as a consultant on software engineering policy and R&D program evaluation to the OECD and the United Kingdom and other governments. Recent books include High Tech Fantasies: Science Parks in Society, Science and Space, Routledge, 1992 (with Darren Massy and David Wield), and Social Dimensions of Systems Engineering: People, Processes, Policies and Software Development. Ellis Horwood, 1993.

Wanlei Zhou received a B.Eng. and an M.Eng. in computer science and engineering from Harbin Institute of Technology, Harbin, China, in 1982 and 1984, respectively. He received a Ph.D. in computer science from the Australian National University, Canberra, Australia, in 1991. He is currently a lecturer in the Department of Information Systems and Computer Science, National University of Singapore (NUS), Singapore. Before joining the NUS, Dr. Zhou was a programmer at Apollo/HP in Massachusetts, U.S.A., and a lecturer at Monash University, Melbourne, Australia. His research interests include distributed computing, software engineering, data communications, performance evaluation, and fault-tolerant computing.
SRPC: A Simple And Fault-Tolerant Remote Procedure Call System

Wanlei Zhou
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract

This paper is concerned mainly with the software aspects of achieving reliable operations on a system of computers that communicate via a network. A system for supporting fault-tolerant, distributed software development is described. The system incorporates fault tolerance features into the RPC system and makes them transparent to users. A buddy is set up for a fault-tolerant server to be its alternative. When an RPC to a server fails, the system will automatically switch to the buddy to seek for an alternate service. Our system is small, simple, easy to use and also has the advantage of producing server and client driver programs and finally receivable programs directly from the server definition files.

Key Words: Fault-tolerant computing, distributed systems, remote procedure calls, client/server model, TCP/IP.

1 Introduction

The advances in computer technology has made it cost-effective to build distributed systems in various applications. A distributed system consists of many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever increasing dependency being placed on distributed systems, the number of users requiring fault tolerance is likely to increase.

This paper is concerned mainly with the software aspects of achieving reliable operations on a system of computers that communicate via a network. The system design is aimed toward application areas in which requirements are less severe than in, for example, the aerospace field, but in which continuous availability are required in the case of some components failures. The application areas could be, for example, supervisory and telecontrol systems, switching systems, process control and data processing. Such systems usually have redundant hardware resources and one of the main purpose of our system is to manage the software redundant resources in order to exploit the hardware redundancy. The fault-tolerant technique used is a variation of recovery blocks [3] and the distributed computing model used is the remote procedure calling (RPC) model.

There have been many successful RPC systems since Nelson’s work[7]. But little of them consider fault tolerance in their design, or they rely on users to build up the fault-tolerant features.

Notable works on incorporating fault tolerance features into RPC systems are the Argus[8], the ISIS[1] and an atomic RPC system on ZMOB[5]. The Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. Atomic transactions are serialisable and indivisible. A user can also define some atomic objects, such as atomic arrays and atomic record, to provide the additional support needed for atomicity. All the user fault tolerance requirements must be specified in the Argus language.

The ISIS toolkit is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. A special process group, called fault-tolerant process group, is established when a group of processes (servers and clients) are cooperating to perform a distributed computation. Processes in this group can monitor one another and can then take actions based on failures, recoveries, or changes in the status of group members. A collection of reliable multicast protocols are used in ISIS to provide failure atomicity and message ordering.

The atomic RPC system implemented on ZMOB uses sequence numbers and calling paths to control the concurrency and atomicity, and uses checkpointing to maintain the ability of recovering from failures. Users do not have to provide synchronisation and recovery themselves; they only need to specify if atomicity is desired. This frees them from managing much complexity.

But when a server (or a guardian in the Argus) fails to function well, an atomic transaction or an atomic RPC has to be aborted in these systems. This is a violation of our continuous computation requirement. The fault-tolerant process groups of the ISIS can cope with process failures and can maintain continuous computation, but the ISIS toolkit is big and relatively complex to use.

In this paper we describe a system, called SRPC (Simple RPC) system, for supporting development of fault-tolerant, distributed software. The SRPC incorporates fault tolerance features into the RPC system and makes them transparent to users. A buddy is set up for a fault-tolerant server to be its alternative. When an RPC
to a server fails, the system will automatically switch to the buddy to seek for an alternate service. The RPC aborts only when both the server and its buddy fail. To obtain this fault-tolerance service, users only need to specify their requirements in a descriptive interface definition language. All the maintenance of fault tolerance is managed by the system in a user-transparent manner. By using our system, users will have confidence on their distributed computing without bothering the fault tolerance details. Our system is small, simple, easy to use and also has the advantage of producing server and client driver programs and finally executable programs directly from the server definition files.

The remainder of this paper is organised as follows: In Section 2, we describe the architecture of the SRPC system. Then Section 3 describes the syntax and semantics of the server definition files and the stub and driver generator. In Section 4, we present an example to show how this system can be used in supporting fault-tolerant, distributed software development. Section 5 is the remarks.

2 System Architecture

The SRPC is a simple and fault-tolerant remote procedure call system [9]. The system is small, simple, expandable and it has facilities supporting fault-tolerant computing. It is easy to understand and easy to use. The SRPC only contains the essential features of an RPC system, such as a location server and a stub generator, among other things. The SRPC system has been used as a distributed programming tool in both teaching and research projects for three years.

The SRPC is implemented by using the Internet socket. It can be used on any BSD-based operating systems, such as BSD4.2, BSD4.3, Ultrix and other similar UNIX systems.

2.1 Server Types

The client/server model is used in the SRPC system. An SRPC program has two parts: a server part and a client part. Usually the server provides a special service or manages an object. The client requests the service or accesses the object by using the remote procedures exported by the server.

There are three types of servers in the SRPC system: simple servers, service providing servers and object managing servers. Figure 1 depicts these three types of servers.

A simple server (Figure 1(a)) is an ordinary server possessing of no fault-tolerant features. When a simple server fails, all RPCs to it have to be aborted.

A service providing server (Figure 1(b)) has a buddy server running somewhere in the network (usually on a host different with the server's), but no communication between the server and its buddy. When a service providing server fails, an RPC to this server will be automatically re-directed to its buddy server by the system. As object changes in the server will not be available in its buddy, a service providing server usually is used in applications such as pure computation, information retrieval (no update), motor-driven (no action memory), and so on. It is not suitable to be used to manage any critical object that might be updated and then shared by clients.

An object managing server (Figure 1(c)) also has a buddy running in the network. It manages a critical object that might be updated and shared among clients. An RPC to such a server, if it will change the object state, is actually a nested RPC. That is, when the server receives such a call from a client, it first checks to see whether the call can be executed successfully (e.g. if the necessary write-locks have been obtained or not). If the answer is no, the call is aborted. If the answer is yes, then the server will call its buddy server to perform the operation as well. When the buddy returns successfully, the call commits (the server and its buddy actually perform the call) and the calling result returns to the client. To ensure the consistency of the objects managed by the server and its buddy, a two-phase commit protocol [2] is used when executing the nested RPC.

Like a service providing server, when an object managing server fails, an RPC to this server will be automatically re-directed to its buddy server by the system.

All buddy servers are simple servers. That means, when a server (service providing or object managing) fails, its buddy server provides alternative service in a simple server manner. Also, when a buddy server fails, a service providing server or an object managing server will be reduced into a simple server.

2.2 The Architecture

The SRPC has the following three components: A Location Server (LS) and its buddy (LS buddy), a System Library, and a Stub and Driver Generator (SDG). This section describes the system architecture from a user's point of view.

From a programmer's viewpoint, after the SDG compilation (see Section 4), the server part of an SRPC program is consisted of a server driver, a server stub, and a file which implements all the remote procedures (called procedure file). The server buddies are transparent to users. The server part (or a server program as it is sometimes called) is a "forever" running program which resides on a host and awaits calls from clients. The client part (or a client program) consists of a client driver and a client stub after the SDG compilation. It runs on a host (usually a different host from the server's host) and makes calls to the server by using the remote procedures exported by the server.

When the client driver makes a call, it goes to the
client stub. The client stub then, through the system library, makes use of the Internet entity of the client host for sending the calling message to the Internet entity of the server's host. At the server host side, the Internet entity will send the calling message to the server stub through the system library. The server stub then reports the call to the server and an appropriate procedure defined in the procedures file is executed. The result of the call follows the calling route reversely, through the server stub, the system library and the Internet entity of the server host, the system library and the Internet entity of the client host, the client stub, back to the client driver. This is called a direct call as the pre-condition of such a call is that the client knows the address of the server before the call.

With the help of the Location Server, the run-time address of a server can be easily accessed. One typical scenario of SRPC programs using LS can be described below: When the server is started, it first registers its location to the LS and then waits for clients' calls. The clients know the server by a name (a character string) defined by the user. When a client is invoked, it consults the LS for the server's location. After the location is found, the client then can make any number of RPCs to that server by using the obtained location (as in a direct call). We name this calling as a typical calling, since most of the time a client does not know server addresses. If a "shutdown" call is issued by a client program, the server un-registers itself from the LS and exits from the system.

Figure 2 depicts the system architecture using a typical RPC. The dashed line represents the RPCs from the user's viewpoint.

2.3 The Location Server

One way of hiding out the implementation details is the use of the Location Server. The LS is used to hide the server locations from users. It maintains a database of server locations and is executed before any other SRPC program is started. After that, it resides on the host and awaits calls from servers and clients.

The Location Server is an object managing server and has a buddy of its own. It has a well-known location, and this location can be easily changed when necessary. A location of a server in the SRPC can be expressed in a triple:

$$(host, port, protocol)$$

where the host is the host name or address on which the server is running, the port is the socket port number of the server and the protocol is the protocol used in client/server communication. The host name and address are assigned by system programmers. The port number in the SRPC can be an integer greater than 5050. It can be assigned by the programmer or the system. The available protocols now are internet_stream and internet_datagram.

The location of the LS buddy server has the same port number and protocol, but has a host address determined at runtime. A maintenance tool is also provided to maintain the location database of the location server. It provides calls to register, to un-register, and to locate an entry of the database. It also can be used to list all the database entries and shutdown the location server. The LS itself is implemented by the SRPC system, using the direct calling method.

Usually there should be one LS (called local LS) running on each host for managing locations of that host, and these local LSs report to the "global LS" (like the NCA/RPC's local and global location brokers[8]). In that case the locations of all LSs can also be hidden from users. We have planned to implement this facility.

2.4 The System Library

One of the advantages of using RPC systems is that the user does not need to know the implementation details of the remote procedure. One only needs to call a pre-defined remote procedure like he or she does to call a local procedure.

The system library is one way of achieving transparency. The library contains all the low-level and system-oriented calls. Its main function is to make the low-level facilities transparent to the upper-level programs. So the stub and driver programs of both server and client will not deal with their Internet entities directly.

The server and client programs must be linked with the system library separately. Reference [9] contains detailed descriptions of the library calls. All the library calls can be divided into the following call levels:

1. SRPC Level: This is the highest level. It contains calls that deal with RPC-related operations.
2. Remote Operation Level: It contains calls that deal with remote operations. These remote operations follow the definitions of the OSI Application level primitives[4].
3. Socket Level: It contains calls that deal with socket level operations.
4. Utility Calls: It contains all the utility calls used in different levels.

The inner most level is the socket level. It deals with socket operations and interfaces with the Internet entity and the UNIX kernel. The remote operation level
provides a uniform interface (similar to the OSI Application level primitives) to the upper RPC system. This makes the SRPC system as portable as possible: only the lower level need be re-programmed when port the SRPC system to other communication protocols. The SRPC level implements all the RPC related calls and provides a user-friendly remote procedure calling interface to application programs. The utility calls provide service calls for different levels.

3 Stub and Driver Generator

3.1 Syntax

The purpose of the stub and driver program generator is to generate stubs and driver programs for server and client programs according to the Server Definition Files (SDF). Next is the syntax of a server definition file:

Listing 1. Server definition file syntax

```gml
<SDF> ::= BEGIN
  <HEADER>
    [ <CONST> ]
  <FUNCS>
END

<HEADER> ::= Server Name: variable
  Comment: string
  Communication Protocol: variable
  [ Client Port: integer ; ]
  [ Server Host: string ]
  [ Server Port: integer ; ]
  [<BUDDY>]

<BUDDY> ::= Buddy <BDTYPE> ; variable
  Using: <LANGUAGE> ;
  ::= Auto | Forced

<BDTYPE> ::= C | Pascal

<CONST> ::= constant

<FUNCTIONS> ::= RPC Functions: RPCS

<RPCS> ::= RPC { <RPC> }

<RPC> ::= Name: string { <PARAM> }

<PARAM> ::= Param: <CLASS> ; declarator

<class> ::= <in> | <out>
```

We use a modified BNF to denote the syntax of definition files. The "variable", "integer", "string", "constant", and "declarator" have the same meanings as in the C programming language. Comments are allowed in the definition file. They are defined the same as in the C programming language (using /* and */).

3.2 Semantics

Most of the descriptions of Listing 1 are self-explanatory. We only highlight the following points:

1. The server's name is defined as a variable in the C language. This name will be used in many places. For example, it is the key in the LS database to store and access server entities. When the client asks the LS to locate a server, it provides the server's name defined here. The name is also used as a prefix in naming all the files generated by the SDG. So two different servers cannot be assigned to the same server name. Otherwise the server who registers to the LS first will be accepted while the server who registers to the LS later will be rejected by the LS.

2. The <BUDDY> part is optional. If it is not specified, the generated server will be a simple server, otherwise it will be a service providing server or an object managing server, according to some definitions in the <RPCS> part (described below). The <BUDDY> part has a buddy definition and a language definition. The buddy definition defines that whether the buddy's name and execution is to be determined by the system (Auto) or to be determined by the programmer (Forced). If Auto is defined, the system will generate the buddy server's name (ServerNameBDy, used for registering and locating it), the buddy's driver and stub files as well as the makefile, and will treat the following variable as the name of the buddy's procedure file. Then, the buddy program will be compiled and executed together with the server program. The host of the buddy program will be determined by the system at run time (with the same port number and protocol as the server if the buddy will be executed on a different host. Otherwise, the port number will be determined by the system).

If Forced is defined, the generator will not generate any buddy's program file and will treat the following variable as the name of the buddy server used for registering and locating. The programming and execution of the buddy server will also be the programmer's responsibility.

The language definition Using defines which language does the buddy program use. The key issue of software fault-tolerant is the design diversity or version independent, and one way of achieving design diversity is through the use of programming languages. If a different language is chosen for each version implementation, then the versions are likely to be more independent, not only due to the diversity of languages, but also because individual language features force programmers toward different implementation decisions. Currently only the C programming language is supported in the SRPC system. We have planned to support the Pascal language implementation soon.

3. The <FUNCS> part defines the remote procedures of the server. At least one remote procedure must be defined. Each remote procedure is defined as a name part and a parameter ( <PARAM> ) part. The name of a remote procedure is simply a variable, with an optional Update definition. The latter definition distinguishes an object managing server with a service providing server. That is, if the <BUDDY> part is defined and the Update is defined in any one RPC definition, the server is an object managing server. If only the <BUDDY> part is defined but no Update part is defined in any RPC definition, the server is a service providing server. The
meaning of the Update definition is: if an Update
is defined following an RPC procedure name, that
procedure must be maintained as a nested RPC af-
flecting both the server and the buddy by the server
program (See Section 2.1).

There can be zero or several parameters in a pro-
cedure, each consisting of a class and a declaration.
The class can be int or out, which tells the SRPC
system that the parameter is used for input or out-
put, respectively. The declaration part is the same
as in the C language. In this version, only simple
character string is allowed in parameter definitions.
Further extensions are under way.

.3 Implementation Issues

After a programmer sends a server definition file to
the generator, the generator first does syntax checking.
No errors are found, several program source files
and makefile are generated. The subsequent process-
ing is specified by the makefile. That is, when using
the make utility, the executable files of both the server
and client will be generated. Figure 3 indicates the structure
of the processing. The dashed lines represent optional
connections.

At least one server definition file must be input to the
SG. If there are more than one server, their SDFs can
be input to the SG simultaneously. If there is only one
SDF file, then the generated client driver can execute
the server's procedures one by one. If the buddy part
is also specified, the generated client can also call the
server procedures directly (this is useful in testing the
server-buddy communication).

If there are more than one SDF file, then for each
server, the SG will generate one set of SDF files, one
set of client files, and one set of buddy files (if the buddy
is defined), respectively. These files are the same as
the servers being processed in single file input described
above. One additional set of client files, the multi-server
program, will also be generated in this case. The client
driver is called a multi-server client driver. It can
execute all the procedures of all the servers one by one. A
future improvement is under way to let the client call
the procedures in parallel.

An Application Example

We use a simple example to show the application of
the SRPC system. Suppose we have a server definition
file called sf.def. It defines a "send-and-forward" sys-

client acts as both message sender and receiver. Next is
the server definition file:

```c
// Store and forward: server definition file */

BEGIN
Server Name: sf;
Comment: Store and forward system;
Communication Protocol: Internet_datagram;
Client Port: 6900;
Buddy Auto: sfBdyOps.c;
Using: C;
#define MXMLNL 64
#define MXMLSGL 600
#define MXMLTRL 80

RPC Functions:
Name: storeMsg Update;
Param: in receiver: char receiver[MXMLNL];
Param: in msg: char msg[MXMLSGL];
Param: out stat: char stat[MXMLTRL];
Name: forwardMsg Update;
Param: in receiver: char receiver[MXMLNL];
Param: out msg: char msg[MXMLSGL];
Name: readMsg;
Param: in char receiver[MXMLNL];
Param: out msg: char msg[MXMLSGL];
Name: listMsg;

END
```

From the header part of this SDF file we know the fol-
lowing: The server is named as sf and the proto-
col used is the Internet_datagram. The server can be
executed on any host using any port assigned by the
system. It will register with the LS at the beginning
of the execution and will un-register from the LS at
the end of the execution. The client can also be exec-
ted on any host but it uses a port number of 6900. A server
buddy is defined and is expected to be established
automatically by the system. The procedure file for the
buddy is sfBdyOps.c and the programming language
used for the buddy is the C language. There are also
three constants defined.

From the RPC functions part we know that four
remote procedures are defined in this SDF file. The
first two RPC functions are marked as Update. So the
server is an object managing server. When the client
calls these two procedures, these two procedures will be
treated as nested calls for maintaining the object con-
istency in both the server and its buddy. The next two
RPC function definitions have no Update marks and
then they will be treated as ordinary RPCs.

When this file is input to the generator, the following
files will be generated:

- af.h Header file, must be included by server,
- its buddy and client drivers and stubs.
After using the make utility (simply use "make" command), three executable files are created:

- sfSer: Server program.
- sfClie: Client program.
- sfBuddy: Server buddy program.

Note that the sfOps.c file only defines the frameworks of the remote procedures (dummy procedures). Their details are to be programmed by the programmer. The sfBuddyOps.c file should be the same as the sfOps.c file (the only possible difference happens when the server buddy uses another programming language such as the Pascal, then the affix of the file would be "paps").

The server driver is simple. It does the initialisation first. Then it registers with the LS and invokes the buddy program on a neighboring host because the buddy is defined as auto in the SDF file. After that it loops "forever" to process incoming calls until the client issues a "shutdown" call. In that case the server un-registers from the LS and exits. The "un-register" call will automatically un-register the buddy from the LS as well. The incoming calls are handled by the server stub and underlying library functions.

The server buddy driver works in the same way as the server program, except that it does not invoke a buddy program. Also the buddy is a simple server and all calls to the buddy will not be nested.

The generated client driver can execute the server's remote procedures one by one. If the server driver is running and the client driver is invoked, the client driver first lists all the remote procedures provided by the server, and asks the user to choose from the list. The following is the menu displayed for this example:

```
Available calls:
  0 s$ShutDown
  1 s$storeMsg(receiver, msg, stat)
  2 s$forwardMsg(receiver, msg)
  3 s$readMsg(receiver, msg)
  4 s$listMsg()
Your choice:
```

After the selection, the input parameters of the named remote procedure are then input from the keyboard. After that, the driver program does some initialisation and the remote procedure is executed and returned results displayed. The actual calling and displaying are handled by the client stub and underlying library functions. The format of all the four RPCs in the client program are the same as the the format listed in the above menu.

The fault-tolerant feature of the system is completely hidden from the user. For this example, all the remote procedure calls from the client program will be first handled by the server. A nested RPC is issued if the incoming call is either s$storeMsg(receiver, msg, stat) or s$forwardMsg(receiver, msg). This is because the two RPCs functions are marked as Update in the SDF file. The nested RPC will ensure that the actions of the incoming call will be made permanent on both the server and its buddy if the call is successful, and no actions of the incoming call will be performed if the call fails. Other two incoming calls, s$readMsg(receiver, msg) and s$listMsg(), will be handled by the server only.

If the server fails (that is, the RPC to the server returns an error), the client program will then send the RPC to the server's buddy.

5 Remarks
A system for supporting fault-tolerant, distributed software development is described in this paper. The system is simple, easy to understand and use, and has the ability of tolerating server failures. It also has the advantage of producing server and client driver programs and finally executable programs directly from the server definition files. The system has been used as a tool of distributed computing in both third year and graduate level teaching, and has been used by some students in their projects.

References
A Performance Evaluation of an Atomic Remote Procedure Call Manager

Wanlei Zhou
Department of Information Systems and Computer Science
National University of Singapore
Lower Kent Ridge Road, Singapore 0511
Email: zhouwl@iscs.nus.sg

Abstract

A performance evaluation model for atomic remote procedure call (RPC) management in a distributed system is presented and analysed. At first, a precise model is built using Markovian chain. As the number of transition states of the precise model increases explosively when the system's scale increases, the precise model is only suitable for very small scale systems. By using some characteristics of the system, a simplified model is built based on the precise model. The simplified model can drastically reduce the number of transition states and maintain very good accuracy. The simplified model can be used to solve medium scale systems comfortably. Further approximation is used to build an approximate model for solving large systems. Finally the results of the analytic models are compared with the simulated and practical results.

Key Words: Distributed Systems; Markovian Chain; Performance Evaluation; Remote Procedure Call (RPC); Atomicity Management.

1 Introduction

There have been some suggestions to maintain the atomicity of a remote procedure call over an object system (a system with a collection of objects, and an object here can be a database file, an entry of a database file, or other proper things). The objectives are to obtain the recoverability and the indivisibility. The recoverability requires that the overall effect of an RPC call is all-or-nothing: either all of the objects addressed by the RPC call remain in their initial states before the RPC call or they are all changed to their final states. The indivisibility requires that the partial effects of an RPC call are invisible to other RPC calls. That is, if the objects being modified by an RPC call are observed over time by a second RPC call, then this second RPC call will either always observe the initial states before the first RPC call, or the final states after the first RPC call.

Linkov and her colleagues consider the atomicity of RPC from the viewpoint of programming languages. Concepts such as guardians, actions, atomic data types, and promises are introduced to ensure the atomicity of some segments of a program. Lin and Gannon [3] have implemented an atomic RPC system on ZMOB (an MIMD system). Sequence numbers and calling paths are used to control the concurrency and atomicity, and checkpointing is used to maintain the ability of recovering from failures. Zhou and Molinari [9] consider the problem from the system's viewpoint by using a runtime manager (called atomic RPC manager) to manage the atomicity of RPC calls. When an RPC call is submitted to the system, the manager will maintain its atomicity by rolling back the RPC call if it is considered failed. A preliminary implementation of the manager has been tested on a network of SUN/HP workstations [2].

Assume that there are N hosts in a distributed system and they are fully connected via a communication network. These hosts are numbered from 1 through N. There are many RPC programs running on these hosts. Each RPC program is a combination of server and client programs. A server program maintains some objects and awaits calls from clients. A client program accesses the objects by calling the server to execute a remote procedure for it. There are one or more atomic RPC managers running on the network, depending on the communication traffic. When an RPC program R1 wants to call another RPC program R2, it sends the calling information to an atomic RPC manager. The manager then calls R2. After R2 completes the remote procedure's execution, it returns the results to the manager. The manager then returns the results to R1. If anything goes wrong during the RPC calling, the atomic RPC manager will take care of the work for maintaining the atomicity. The time that R1 spends from the is-
suing of the call to the arriving of the results is called the \textit{calling time}. While the time interval that \( R_i \) issues the next call to any other RPC programs after the previous one is called the \textit{request interval time}. The call is queued if all the managers are occupied. Figure 1 depicts a simplified calling semantics of the system. Note that an atomic RPC manager can locate on any host.

For convenient description, we give the following definition:

\textbf{Definition 1.} A system of \( M \) RPC programs and \( A \) atomic RPC managers is called an \( M \times A \) system.

To analyse this system, we distinguish the following states for RPC programs. In an \( M \times A \) system, an RPC program \( R_i \) can be in one of the following states:

1. \textit{Private}. \( R_i \) does not call any other RPC programs. It only processes its own task.

2. \textit{Shared}. \( R_i \) is calling \( R_j \) or \( R_i \) is called by \( R_j \). They occupy an atomic RPC manager during the call.

3. \textit{Waiting}. \( R_i \) is going to call \( R_j \) but all the atomic RPC managers are occupied or \( R_j \) is in shared state. \( R_i \) is then put into a First-Come-First-Served (FCFS) waiting queue.

In this system, the atomic RPC managers can form a bottle neck problem if there are too many RPC programs or the calling traffic among RPC programs are too heavy. Of course RPC programs are also targets of competition, but this will happen only after a manager is occupied. As we usually have \( M \gg A \), the competition for managers could be remarkable. If \( A \geq \left\lfloor \frac{M}{2} \right\rfloor \) (i.e. the integer part of \( \frac{M}{2} \)), there will be no conflict for atomic RPC managers. The competition may happen if \( A < \left\lfloor \frac{M}{2} \right\rfloor \). Also, under the same condition, the greater \( \left\lfloor \frac{M}{2} \right\rfloor - A \) is, the larger the probability of conflict is. The situation of \( A \geq \left\lfloor \frac{M}{2} \right\rfloor \) is of little interest, unless we want to serve other purposes (such as fault-tolerance). We will assume \( A < \left\lfloor \frac{M}{2} \right\rfloor \) from now on.

2 \textbf{Assumptions and Performance Measures}

We have the following assumptions:

1. The private state is an RPC program’s primary state. An RPC program starts from this state and returns to this state after the shared state.

2. The only way of RPC calling between two RPC programs is through an atomic RPC manager.

3. An RPC program does nothing but waits after it enters the waiting state, unless another RPC program forces it to go into the shared state or it enters the shared state by occupying an atomic RPC manager.

4. Each RPC program has the same mean calling time. That is, \( \forall i \in \{1, \cdots, M\} \), \( R_i \)'s calling time is an exponentially distributed random variable with parameter \( \mu \). So \( \frac{1}{\mu} \) is the mean calling time.

5. Each RPC program has the same request interval time. That is, \( \forall i \in \{1, \cdots, M\} \), \( R_i \)'s request interval time is an exponentially distributed random variable with parameter \( \lambda \). So \( \frac{1}{\lambda} \) is the mean request interval time.

6. Each RPC program has the same probability of calling another RPC program. That is, \( \forall i, j \in \{1, \cdots, M\}, i \neq j \), the probability that \( R_i \) calls \( R_j \) is \( p_{ij} = \frac{1}{M-1} \).

Based on the above assumptions, we can build the analytic model for some very simple systems and solve the associated performance problems. From the above assumptions, we know that all RPC programs
are symmetric and all atomic RPC managers are also symmetric. This property is very useful during the model's simplification.

We use the performance metrics of queueing theory to describe the structure and performance measures of our model. In a queueing system, usually we want to know the mean response time (from the point that a customer enters the system to the point that it leaves the system) \( W \), a customer’s waiting time in the system \( W_q \), the mean number of customers in the queuing system \( L \), the mean queueing length \( L_q \), and so on. Other performance measures can be derived from the above measures [1].

However, it is not so easy to obtain so many performance measures simultaneously. In fact we can relate all these performance measures to a system index called processing power [7]. We can first obtain this index, then use this to obtain other performance measures.

**Definition 2.** System processing power is defined as:

\[
P = \text{the expectation of number of private state RPC programs in the system.}
\]

\[
P = \lim_{k \to \infty} k \cdot p_k
\]  \hspace{1cm} (1)

where \( p_k \) is the probability of \( k \) private state RPC programs in the system.

The relation between \( P \) and other performance measures follows:

Let \( \Lambda \) be the rate of RPC programs cycling through the system. As \( \lambda \) is the request rate of one private state RPC program, and \( P \) is the number of private state RPC programs, we have

\[
\Lambda = P \cdot \lambda.
\]  \hspace{1cm} (2)

Let \( M \) be the number of RPC programs in the whole system. Obviously, the mean number of RPC programs in the queueing system (the mean number of RPC programs that are in shared and waiting states) is

\[
L = M - P
\]  \hspace{1cm} (3)

The mean number of RPC programs waiting in the queue can be obtained by \( L \) minus the mean number of RPC programs in shared state \( \frac{\Lambda}{\mu} \):

\[
L_q = L - \frac{\Lambda}{\mu} = M - P(1 + \rho)
\]  \hspace{1cm} (4)

where \( \rho = \frac{\Lambda}{\mu} \), and equations (2) and (3) are used in the second step. From Little’s law we have:

\[
L = \lambda \cdot W
\]  \hspace{1cm} (5)

\[\begin{align*}
\lambda &\quad (0, 0, 0) \\
\mu &\quad (2, 1, 0) \\
\lambda &\quad (3, 0, 1) \\
\mu &\quad (0, 3, 2) \\
\end{align*}\]

Figure 2: 3 * 1 system state transition diagram (precise model)

where \( W \) is the mean response time. So,

\[
W = \frac{L}{\lambda} = \frac{M - P}{P \lambda}
\]  \hspace{1cm} (6)

The mean queuing time can be obtained from \( W \) with the deduction of the mean calling time \( \frac{1}{\mu} \):

\[
W_q = W - \frac{1}{\mu} = \frac{M - P(1 + \rho)}{P \lambda}
\]  \hspace{1cm} (7)

So, in the following discussions, we will concentrate on processing power \( P \).

### 3 Precise Model

Precise model is relatively more precise than the later simplified and approximate models. The state of this model is defined as an \( M \)-tuple for an \( M \times A \) system:

\[
(S_1, S_2, \ldots, S_M)
\]

where \( S_i \) is the state of RPC program \( R_i \). It has the value of:

\[
S_i = \begin{cases} 
0 & R_i \text{ is in private state} \\
\begin{array}{c}
k \\
(\text{shared state})
\end{array} & R_i \text{ is calling or called by } R_k \\
-t & R_i \text{ is in waiting state, and in } k\text{th position of the queue}
\end{cases}
\]

**Example 1.** Use precise model to solve 3*1 system.

**Solution:** Figure 2 is the state transition diagram of the 3 * 1 system.
When there are no private state RPC programs (that is, $S_i \neq 0$ for all $S_i$'s of formula (8)), the rate of request for calling RPC programs will be 0. So, the system will be in a stable state and we can use the stable-state equations (input stream = output stream) to obtain $p_0 (i = 0, 1, 2, 3)$. Then by using formula (1), we can obtain the processing power of a $3 \times 1$ system:

$$P_{3\times1} = \frac{3(3\rho + 1)}{3\rho^2 + 6\rho + 1}$$ (9)

where $\rho = \frac{\lambda}{\mu}$.

However, when the number of RPC programs and the number of atomic RPC managers increase, the number of transition states of the precise model increases explosively. In fact, it is already very difficult to draw the state transition diagram of a $5 \times 1$ system, needless to say to solve it by hand. Let us look at how the number of states changes in the precise model.

**Corollary.** The number of queueing methods for $N$ RPC programs is

$$D_N = \sum_{k=0}^{N} \frac{N!}{(N-k)!}$$

**Proof:** A queueing method of $N$ RPC programs means that any $k$ (0 $\leq k \leq N$) RPC programs form a queue, and they are in positions 1, 2, ..., $k$, or the queue can be empty ($k = 0$). While other ($N - k$) RPC programs are not in the queue.

Evidently, there are $C_N^k$ methods to select $k$ RPC programs from $N$ RPC programs, and there are $k!$ methods to queue these $k$ RPC programs. So the number of queueing methods for length $k$ queues is $C_N^k \cdot k!$. And the whole number of queueing methods for $N$ RPC programs is

$$\sum_{k=0}^{N} C_N^k \cdot k! = \sum_{k=0}^{N} \frac{N!}{(N-k)!}$$

**Theorem 1.** The number of transition states of an $M \times 1$ system is:

$$T(M, 1) = 1 + C_M^2 \cdot D_{M-2} + \sum_{k=1}^{M-2} C_M^k \cdot k!$$ (10)

**Proof:** For an $M \times 1$ system there can be at most two RPC programs in shared state because there is only one atomic RPC manager. So there are $C_M^2$ methods to select these two RPC programs. After that, other ($M - 2$) RPC programs can only be in private or waiting states. That is, they can form a queue. According to our Corollary, there are $D_{M-2}$ different queues for them. So the number of transition states for this situation is $C_M^2 \cdot D_{M-2}$.

Another situation is that when the two shared state RPC programs release the atomic RPC manager and return to private states. Now the manager is not occupied and the waiting queue can be empty or not empty. There is only one transition state that all $N$ RPC programs are in private state and then the queue is empty. If the queue is not empty, then from our Corollary, the number of transition states for this situation is $\sum_{k=1}^{M-2} C_M^k \cdot k!$. So the whole number of transition states is:

$$T(M, 1) = 1 + C_M^2 \cdot D_{M-2} + \sum_{k=1}^{M-2} C_M^k \cdot k!$$

**Theorem 2.** The number of transition states of an $M \times A$ system is:

$$T(M, A) = T(M, 1) + \sum_{k=1}^{A-1} C_M^{(k+1)} \cdot D_{M-2(k+1)}$$ (11)

**Proof:** When $A = 1$, equation (11) holds because of Theorem 1. Suppose the theorem holds for $A = n - 1$, then when $A = n$ we have:

$$T(M, n) = T(M, n-1) + C_M^n \cdot D_{M-2n}$$ (12)

The meaning of equation (12) is: when one more atomic RPC manager is added into the system (n is increased by 1), $T(M, n)$ will include all the number of transition states before this increment ($T(M, n-1)$), and the increased number of transition states. The latter portion is formed by the following situation: there are 2n RPC programs in shared state and other ($M - 2n$) RPC programs are in different queueing conditions. From the Corollary we know this is $C_M^{2n} \cdot D_{M-2n}$. Hence, by the principle of mathematical induction,

$$T(M, A) = T(M, 1) + \sum_{k=1}^{A-1} C_M^{(k+1)} \cdot D_{M-2(k+1)}$$
Table 1 lists the explosive increasing of transition states for the precise model as $M$ and $A$ increase.

Table 1. The increment of transition states for precise model

<table>
<thead>
<tr>
<th>$M$</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>10</th>
<th>4</th>
<th>5</th>
<th>6</th>
</tr>
</thead>
<tbody>
<tr>
<td>$A$</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>$T$</td>
<td>2</td>
<td>10</td>
<td>47</td>
<td>246</td>
<td>7,542,596</td>
<td>48</td>
<td>256</td>
<td>1,597</td>
</tr>
</tbody>
</table>

So the precise model is not suitable for practical calculation. We need to simplify it. The next two sections are devoted to the model simplification.

4 Simplified Model

Two reasons cause the number of transition states to increase intolerably. First, we have to record in each transition state the queuing and calling situations of each RPC program. That means that each element of tuple (8) can have many possible values. Permuting all these possible values will form many transition states. Second, we do not make use of the symmetric property of this system. As we mentioned previously, all the RPC programs are treated equally and they have the same request rate and mean calling time. So each RPC program has the same property from the viewpoint of statistics. As soon as we know the collective property of all the RPC programs, we know the property of individuals. The simplified model simplifies the precise model from these two factors. The state definition of a simplified model is:

$$ (N, K) $$ \hspace{1cm} (13)

where $N$ is the number of pairs of RPC programs in shared state, and $K$ is the number of RPC programs in private state. So $M - 2(N + K)$ is the number of RPC programs in waiting state. $N$ and $K$ in formula (13) can have the following values:

$$ N = \begin{cases} 0 & \text{no RPC programs in shared state} \\ n & \text{n pairs of RPC programs in shared state} \end{cases} $$

$$ K = \begin{cases} 0 & \text{no RPC programs in private state} \\ k & \text{k RPC programs in private state} \end{cases} $$

and $0 \leq n \leq A$, $0 \leq k \leq M$ for an $M \times A$ system.

As in the precise model situation, some of the value combinations are not allowed (for example, we must have $2n + k \leq M$).

We use $S_{ij}$ to denote state $(i, j)$, i.e., there are $i$ pairs of shared state RPC programs and $j$ private state RPC programs. Next we compare the number of transition states of the simplified model with that of the precise model. Then we will analyze the transition rates of our simplified model.

Theorem 3. The number of transition states of a simplified $M \times A$ system is:

$$ T(M, A) = (A + 1)M - (A^2 + 1) $$ \hspace{1cm} (14)

Proof: We use the enumeration method to prove this theorem. This method can help us to order transition states.

Assume at first that there are shared state RPC programs. When there is one pair of shared state RPC programs, the other $(M - 2)$ RPC programs can form the following $(M - 2 + 1)$ transition states:

$$ S_{10}, S_{11}, S_{12}, \ldots, S_{1(M-2)} $$

These states represent the situations that there is one pair of shared state RPC programs and $k$ private state RPC programs, $k = 0, 1, \ldots, (M - 2)$.

When there are two pairs of shared state RPC programs, the other $(M - 2 \times 2)$ RPC programs can form the following $(M - 2 \times 2 + 1)$ transition states:

$$ S_{20}, S_{21}, S_{22}, \ldots, S_{2(M-2\times2)} $$

Situations for other number of pairs of shared state RPC programs can be derived in the same way. The maximum number of pairs of shared state RPC programs is $A$, so the other $(M - 2 \times A)$ RPC programs can form the following $(M - 2A + 1)$ transition states:

$$ S_{A0}, S_{A1}, S_{A2}, \ldots, S_{A(M-2A)} $$

Another situation is when there are no shared state RPC programs. This includes situations that all RPC programs are in private state and when a pair of shared state RPC programs returned to private state while other queuing RPC programs have not entered the shared state yet. The following $(M - 1)$ transition states represent this situation:

$$ S_{00}, S_{01}, S_{02}, \ldots, S_{0M} $$

So the total number of transition states is:

$$ T(M, A) = (M - 1) + \sum_{i=1}^{A} (M - 2i + 1) $$

$$ = (M - 1) + AM - A^2 $$

$$ = (A + 1)M - (A^2 + 1). $$
As in Table 1, we also list the increasing of transition states for simplified model as \( M \) and \( A \) increase.

**Table 2. The increment of transition states for simplified model**

<table>
<thead>
<tr>
<th>( M )</th>
<th>2</th>
<th>3</th>
<th>4</th>
<th>5</th>
<th>10</th>
<th>4</th>
<th>5</th>
<th>6</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>( A )</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>1</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td>2</td>
<td></td>
</tr>
<tr>
<td>( T )</td>
<td>2</td>
<td>4</td>
<td>6</td>
<td>8</td>
<td>18</td>
<td>7</td>
<td>10</td>
<td>13</td>
<td>25</td>
</tr>
</tbody>
</table>

Compare Table 2 with Table 1 we know that the simplified model reduces the number of transition states drastically. This makes the medium-sized systems solvable.

Next let us derive the transition rates for the simplified model. We use \( S_{ij} \rightarrow S_{kl} \) to denote the state transition from \( S_{ij} \) to \( S_{kl} \), and use \( R(S_{ij} \rightarrow S_{kl}) \) to denote its transition rate.

Suppose we are now in state \( S_{ij} \). From the state transition definition and Markovian model, only the following five transfers are possible:

1. \( S_{ij} \rightarrow S_{i+1,j-2} \): It means that when there are \( i \) pairs of shared state RPC programs and \( j \) private state RPC programs, the transition of one private state RPC program calls another private state RPC program and both enter the shared state. There are \( j \) methods for selecting the caller and \( (j - 1) \) methods for selecting the callee. While we know from Section 2 that the probability of one RPC program calls another is \( \frac{\lambda}{M-1} \), so

\[
R(S_{ij} \rightarrow S_{i+1,j-2}) = j(j-1)\frac{\lambda}{M-1}
\]

And the conditions for this transition are \( i+1 \leq A, j-2 \geq 0 \) and \( -(i = 0) \land (2i + j < M) \). These conditions represent that there are idle atomic RPC managers, there are at least 2 private state RPC programs and exclude the situation that there is no shared state RPC programs and the waiting queue is not empty.

2. For \( S_{ij} \rightarrow S_{i+1,j-1} \): It means that when there are \( j \) private state RPC programs and \( (M-2i-j) \) waiting state RPC programs, the transition of one waiting state RPC program calls a private state program, or one private state RPC program calls a waiting state RPC program and both enter the shared state. The first situation has \( j \) selections because only the first of the queue can select (the queue is assumed FCFS in Section 2). The latter is the situation of private state RPC programs call waiting state RPC programs and has \( j(M-2i-j) \) selections. So

\[
R(S_{ij} \rightarrow S_{i+1,j-1}) = j(M-2i-j)\frac{\lambda}{M-1}
\]

And the conditions for this transition are \( 0 < i < A \) and \( j-1 > 0 \). If \( i = 0 \) then only the first situation can occur, i.e., the first of the queue calls one of the private RPC programs:

\[
R(S_{ij} \rightarrow S_{i+1,j-1}) = j\frac{\lambda}{M-1}, \quad (j \geq 1) \land (i = 0)
\]

3. For \( S_{ij} \rightarrow S_{i+1,j} \): This is the transition of one waiting state RPC program calls another waiting state RPC program after a pair of shared state RPC programs returned to private states. And there must be \( i+1 = A \), i.e., the waiting state RPC program entered that state because there was no atomic RPC managers available. There is only one method to select the caller: the first of the queue, while the selection methods for the callee can be \( (M-2i-j-1) \). So

\[
R(S_{ij} \rightarrow S_{i+1,j}) = (M-2i-j-1)\frac{\lambda}{M-1}
\]

And the conditions are \((i+1 = A) \land (M-2i-j \geq 2)\).

4. For \( S_{ij} \rightarrow S_{i,j-1} \): Two situations should be distinguished here: the first is that there are available atomic RPC managers \((i < A)\) and there are shared state RPC programs \((i > 0)\). This means a transition of one private state RPC program calls a shared state RPC program while there are \( i \) private state RPC programs. The second is that there is no available atomic RPC managers \((i = A)\). This means a transition of a private state RPC program calls one of the other RPC programs when there are \( i \) private state RPC programs.

\[
R(S_{ij} \rightarrow S_{i,j-1}) = \begin{cases} 2ij\frac{\lambda}{M-1} & (0 < i < A) \\ j\lambda & (i = A) \land (j \geq 1) \end{cases}
\]

5. For \( S_{ij} \rightarrow S_{i-1,j+2} \): This means the transition of a pair of shared state RPC programs return to private states when there are \( i \) pairs of shared state RPC programs. So

\[
R(S_{ij} \rightarrow S_{i-1,j+2}) = i\mu
\]

And the conditions are \((i > 0) \land (j + 2 \leq M)\).
Example 2. Use simplified model to solve 3*1 system.

Solution: Figure 3 depicts the state transition diagram.

It is easy to obtain the stable-state equations from the transition diagram (also use input stream = output stream). By solving the equations we can obtain: \( p_i(i = 0, 1, 2, 3) \). And then by using formula (1), we can obtain the processing power of the simplified 3*1 system:

\[
P_{3*1} = \frac{3(3\rho + 1)}{8\rho^2 + 6\rho + 1}
\]

where \( \rho = \frac{\lambda}{\mu} \).

Comparing result (15) with result (9), one can see that the processing power for simplified 3*1 model is the same as the precise 3*1 model. This is because we use the symmetric property to do the simplification. We view all the RPC programs and atomic RPC managers equally and consider their effect collectively instead of individually. So one transition state in the simplified model in fact represents some of the transition states of the precise model. For instance, the state (1, 0) in Figure 3 represents states (2, 1, -1), (3, -1, 1) and (-1, 3, 2) in Figure 2. It represents the state that there is one pair of shared state RPC programs, no private state RPC program, and 3 - 2 x 1 = 0 = 1 waiting state RPC program. From Figure 2 and the solution of Example 1, we know that the \( p_k \)'s of these three states are equal. And their sum is exactly the value of \( p_{(1,0)} \). Of course if there is no symmetric property, the simplified model is only an approximation of the precise model.

5 Approximate Model

Although we can solve medium-sized systems by using the simplified model, for large-sized systems this model is still difficult to solve. The approximate model of this section can be used to solve large-size systems.

The main reason that the simplified model still quickly increases its number of transition states is because that we considered the effect of the waiting queue (eg. the RPC program in the first position of the queue). The approximate model will not consider a waiting queue any more. It only distinguishes the number of RPC programs in waiting and shared states and the number of RPC programs in private state. This method can not reflect the whole characteristics of the system. It can, however, greatly reduce the number of transition states. We will see later that this method also has reasonable accuracy.

The state definition for approximate model is:

\( V \)

where \( V \) is the number of RPC programs in private state. Evidently \( M - V \) is the number of RPC programs of shared and waiting states. It is easy to know that the number of transition states of an \( M*A \) system is \( (M + 1) \). This number is greatly less than that of the simplified model, especially when \( M \) and \( A \) are large.

Now we can analyse the transition rates. Let \( S_i \) be the state of \( i \) private RPC programs, \( S_i \rightarrow S_j \) be the transition from state \( S_i \) to state \( S_j \) and \( R(S_i \rightarrow S_j) \) be this transition rate.

If the current state is \( S_i \), then the possible state transitions are:

\[ S_i \rightarrow S_{i-2}, \quad S_i \rightarrow S_{i+2}, \quad S_i \rightarrow S_{i-1} \]

We now analyse them one by one.

1. \( S_i \rightarrow S_{i-2} \): This concerns the transition of a pair of RPC programs from a private state to a shared state when there are \( i \) private state RPC programs in the system. In order to have this transition, there must be available atomic RPC managers. But there is no such specification in the model. So we can not express this state transition exactly. Through experiment, however, we think a constant factor can be added into the transition rate expression, i.e.:

\[ R(S_i \rightarrow S_{i-2}) = K \cdot (i - 1) \frac{\lambda}{M - 1} \]

The conditions are: \( 2 \leq i \leq M - 2 \) and \( A > 1 \). Currently we let \( K = 0.86 \).

When \( A = 1 \), i.e., there is only one atomic RPC manager, then the only possible transition is \( S_M \rightarrow S_{M-2} \), and

\[ R(S_M \rightarrow S_{M-2}) = M(M - 1) \frac{\lambda}{M - 1} = M \lambda \]

101
Also if \( i = M \), the transition rate is the same as above. So

\[
R(S_i \rightarrow S_{i-2}) = \\
\begin{cases} \\
K \cdot i(i-1) \lambda_i^{M-i}, & (2 \leq i \leq M-2) \wedge (A > 1) \\
M \lambda, & (A = 1) \lor (i = M) \\
\end{cases}
\]

2. \( S_i \rightarrow S_{i+2} \): It represents the transition of a pair of shared state RPC programs returns to their private states. There are two situations. If \( M - i \geq 2A \) (i.e., the number of shared state and waiting state RPC programs is greater than the number of needed RPC programs to occupy all the atomic RPC managers), then all the atomic RPC managers may be occupied already. If \( M - i < 2A \), then there are available atomic RPC managers and the number of occupied atomic RPC managers is at most \( M_i = \lfloor \frac{M-i}{2} \rfloor \). Of course we must have \( i + 2 \leq M \) as the condition. So

\[
R(S_i \rightarrow S_{i+2}) = \\
\begin{cases} \\
A \cdot \mu, & (M - i \geq 2A) \wedge (i + 2 \leq M) \\
M_i, & (M - i < 2A) \wedge (i + 2 \leq M) \\
\end{cases}
\]

3. \( S_i \rightarrow S_{i-1} \): This is the most complex situation. We consider it by four classes. Each class considers the system’s one behaviour in the simplified model and obtain its average. The result follows:

\[
R(S_i \rightarrow S_{i-1}) = \\
\begin{cases} \\
\lambda, & (i = 1) \lor ((A = 1) \wedge (i = M - 1)) \\
(2A \cdot \lambda M_i - 1 + \lambda) / 2, & (A > 1) \wedge (i = M - 1) \\
\lambda(1 - (M - i - 1) + (M - i) + (M - i - 2A) / 2A(M - 1)), & M - i < 2A \\
\lambda(1 - (M - i - 1) + (M - i - 2A) / 2A(M - 1)), & M - i \geq 2A \\
\end{cases}
\]

It is convenient to solve a large system by using the approximate model. But using it to solve small scale systems is not necessary. However, for completeness we show the solution for 3 * 1 system using the approximate model here. The state transition diagram is depicted in Figure 4.

We can see that this diagram is similar to Figure 3 of the simplified model. Actually the only difference is the names of the states. The processing power is also the same, that is, formula (15). Of course the results for 4 * 1 system will be different.

6 Results Analysis

The following topics are analysed:

- Compare calculated results of simplified and approximate models with results of a simulation system.
- Compare the results between simplified model and approximate model.
- Analyse the effect of increasing the number of atomic RPC managers to the system’s performance.

![Figure 4: 3 * 1 system state transition diagram (approximate model)](image)

<table>
<thead>
<tr>
<th>Load</th>
<th>Measured</th>
<th>Simplified</th>
<th>Approx.</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \rho = \lambda / \mu )</td>
<td>( P )</td>
<td>( P )</td>
<td>( P )</td>
</tr>
<tr>
<td>0.01</td>
<td>3.850</td>
<td>3.886</td>
<td>3.856</td>
</tr>
<tr>
<td>0.1</td>
<td>3.165</td>
<td>3.171</td>
<td>3.195</td>
</tr>
<tr>
<td>0.2</td>
<td>2.808</td>
<td>2.815</td>
<td>2.857</td>
</tr>
<tr>
<td>0.625</td>
<td>2.019</td>
<td>2.032</td>
<td>2.077</td>
</tr>
<tr>
<td>1.0</td>
<td>1.607</td>
<td>1.629</td>
<td>1.647</td>
</tr>
<tr>
<td>1.4</td>
<td>1.301</td>
<td>1.317</td>
<td>1.326</td>
</tr>
<tr>
<td>2.0</td>
<td>1.001</td>
<td>1.015</td>
<td>1.007</td>
</tr>
</tbody>
</table>

First we have implemented a real atomic RPC manager system on a network of SUN/HP workstations. We let one atomic RPC manager run on the system and use four RPC programs periodically calling each other (i.e., a 4 * 1 system). Then we measure \( p_k \) (\( k = 0, 1, 2, 3, 4 \)) on the atomic RPC manager by using different system loads (\( \rho = \frac{\lambda}{\mu} \) changes) and calculate the processing power \( P \). Two machines are involved in the execution. They are isolated from the rest of the network to reduce interruption. Altogether 1000 times of execution and measurement are carried out and the average values of \( p_k \)'s are used in our calculation. After that we use the simplified and approximate models to solve the 4 * 1 system for these loads. Table 3 lists the results of this comparison. We can see that the simplified method and approximate method produce similar results as the measured method.

For better understanding of the analytic models, we have implemented a simulation program for
$M \times A$ systems. The simulation language used is an extended version of the siml discrete-event simulation language[6]. Then we compare the results of the simplified and the approximate models with the simulated results. The comparison shows that the results of analytic models are close to the results of simulated method. The difference is within 10% range for most of the systems.

The third aspect of our analysis is on the comparison of the simplified model and the approximate model. We use these two models to calculate the results for $3 \times 1, 4 \times 1, \ldots$ till $12 \times 2$ systems (altogether 20 systems) and calculate the differences of these two analytic systems. Figure 5 gives the absolute differences of the two models. It shows that the absolute differences are large when $M \leq 8$. When $M$ and $A$ increase, the absolute differences decrease. So the analytic models are not suitable for smaller systems (except $3 \times 1$ and $4 \times 1$ systems).

The next research topic is about the relationship of $A$ (number of atomic RPC managers) and $P$ (system processing power). We use the approximate model to calculate the results for some systems when fixing $M$ and changing $A$. Figure 6 depicts the processing powers of $8 \times 1, 8 \times 2, 8 \times 3$ and $8 \times 4$ systems. It shows that the system processing power $P$ increases when the number of atomic RPC managers increases (so the waiting time and length of queue will decrease). But the increment of $P$ reduces when $P$ reaches some value. This result matches our intuition. So it is necessary to select a proper number of atomic RPC manager for a given number of RPC programs in the system.

7 Remarks

The performance evaluation of an atomic RPC manager system is described in this paper. Markov chains are used in the analyse, and models of precise, simplified and approximate natures are given. Each model has its suitable applications. The precise model can describe a system precisely but the number of transition states of the model increases explosively when the number of atomic RPC managers or the number of RPC programs increase. This limits the precise model to be used in very small systems. However the theory of the precise model is an important basis for other models.

The simplified model makes use of the symmetric characteristic of the system. It drastically reduces the number of transition states and maintains very good accuracy. The simplified model is suitable for medium-sized systems. For a large system, the number of transition states can still be unmanageable.

The approximate model is developed to deal with large systems. We ignore many differences between a shared state and a waiting state in this model and the resultant model can deal with large systems easily with reasonable accuracy.

References


ence, National University of Singapore, January 1993.


A Fault-Tolerant Remote Procedure Call System for Open Distributed Processing

Wanlei Zhou, School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia

This paper is concerned mainly with the software aspects of achieving reliable operations on an open distributed processing environment. A system for supporting fault-tolerant and cross-transport protocol distributed software development is described. The fault-tolerant technique used is a variation of the recovery blocks and the distributed computing model used is the remote procedure call (RPC) model. The system incorporates fault tolerance features and cross-transport protocol communication features into the RPC system and makes them transparent to users. Our system is small, simple, easy to use and also has the advantage of producing server and client driver programs and finally executable programs directly from the server definition files.

Keywords Codes: C.2.4, D.4.4, D.4.5.

Keywords: Open distributed processing, Fault-tolerant computing, distributed systems, remote procedure calls, client/server model.

1 Introduction

The advances in computer technology has made it cost-effective to build distributed systems in various applications. Many experts agree that the future of open distributed processing is the future of computing. The network is the computer has become a popular phrase [5].

Remote Procedure Call (RPC) is perhaps the most popular model used in today's distributed software development and has become a de facto standard for distributed computing. To use it in an open distributed environment effectively, however, one has to consider the cross-protocol communications because user programs built on top of different RPC systems cannot be interconnected directly. Typical solutions to this problem are:

1. Black protocol boxes: protocols used by RPC programs are left as black boxes in compiling time, and are dynamically determined in binding time [1].


However, one issue is still outstanding in building RPC systems for open distributed systems: the fault-tolerance features.

An open distributed system consists of many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever increasing dependency being placed on open distributed systems, the number of users requiring fault tolerance is likely to increase.

This paper is concerned mainly with the software aspects of achieving reliable operations on an open distributed processing environment. A system for supporting fault-tolerant and cross-transport protocol distributed software development is described. The

---

1 Proceedings of the 1995 IFIP International Conference on Open Distributed Processing, pp. 229-246, Brisbane, Queensland, February 21-24, 1995
system design is aimed toward application areas that may involve heterogeneous environment and in which requirements for fault-tolerance are less severe than in, for example, the aerospace field, but in which continuous availability are required in the case of some components failures [4]. The application areas could be, for example, kernel/service pool-based distributed operating systems, supervisory and telecontrol systems, switching systems, process control and data processing. Such systems usually have redundant hardware resources and one of the main purpose of our system is to manage the software redundant resources in order to exploit the hardware redundancy.

The reminder of this paper is organised as following: In Section 2, we summary some notable related work provide the rationale of our work. In Section 3, we describe the architecture of the SRPC system. Then Section 4 describes the syntax and semantics of the server definition files and the stub and driver generator. In Section 5, we present an example to show how this system can be used in supporting fault-tolerant, open distributed software development. Section 6 is the remarks.

2 Related Work And The Rationale

There have been many successful RPC systems since Nelson's work [11]. But few of them consider fault tolerance and cross-protocol communication in their design, or they relay on users to build up these features.

Notable works on incorporating fault tolerance features into RPC systems are the Argus [10] and the ISIS [2] [3]. The Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. Atomic transactions are serialisable and indivisible. A user can also define some atomic objects, such as atomic arrays and atomic record, to provide the additional support needed for atomicity. All the user fault tolerance requirements must be specified in the Argus language.

The ISIS toolkit is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. A special process group, called fault-tolerant process group, is established when a group of processes (servers and clients) are cooperating to perform a distributed computation. Processes in this group can monitor one another and can then take actions based on failures, recoveries, or changes in the status of group members. A collection of reliable multicast protocols are used in ISIS to provide failure atomicity and message ordering.

However, when a server (or a guardian in the Argus) fails to function well, an atomic transaction or an atomic RPC has to be aborted in these systems. This is a violation of our continuous computation requirement. The fault-tolerant process groups of the ISIS can cope with process failures and can maintain continuous computation, but the ISIS toolkit is big and relatively complex to use.

Typical solutions to the cross-protocol communication in RPC systems are the black protocol boxes of the HRPC [1], the special protocol conversion interface [15] and the RPC agent synthesis system [7] for cross-RPC communications.

The HRPC system defines five RPC components: the stub, the binding protocol, the data representation, the transport protocol, and the control protocol. An HRPC client or server and its associated stub can view each of the remaining components as a "black box." These black boxes can be "mixed and matched." The set of protocols to be used is determined at bind time – long after the client and server has been written, the stub has been generated, and the two have been linked.

The special protocol conversion interface proposed in [15] uses an "interface server" to receive a call from the source RPC component (client or server) and to convert it into the call format understood by the destination RPC component (server or client).

The cross-RPC communication agent synthesis system proposed in [7] associates a "client agent" with the client program and a "server agent" with the server program. A "link protocol" is then defined between the two agents and allow them to communicate. The server and the client programs can use different RPC protocols and the associated
agents will be responsible of converting these dialect protocols into the link protocol.

But none of the above cross-protocol RPC systems consider fault-tolerance issues. If
the server fails, the client simply fails as well.

Incorporating both fault tolerance and cross-protocol communication into RPC systems
is clearly an important issue for using RPCs efficiently and reliably in open distributed
environments. In this paper we describe a system, called SRPC (Simple RPC) system,
for supporting development of fault-tolerant, open distributed software. The SRPC incor-
porates fault tolerance features and protocol converters into the RPC system and makes
them transparent to users. A buddy is set up for a fault-tolerant server to be its alter-
native. When an RPC to a server fails, the system will automatically switch to the buddy to
seek for an alternate service. The RPC aborts only when both the server and its buddy fail.
The clients and servers can use different communication protocols. To obtain these
fault tolerance and automatic protocol converting services, users only need to specify their
requirements in a descriptive interface definition language. All the maintenance of fault
tolerance and protocol conversion are managed by the system in a user transparent man-
ner. By using our system, users will have confidence on their open distributed computing
without bothering with the fault tolerance details and protocol conversion. Our system is
small, simple, easy to use and also has the advantage of producing server and client driver
programs and finally executable programs directly from the server definition files.

3 System Architecture

The SRPC is a simple, fault-tolerant and cross-protocol remote procedure call system
[16]. The system is small, simple, expandable and it has facilities supporting fault-tolerant
computing and cross-protocol communication. It is easy to understand and easy to use.
The SRPC only contains the essential features of an RPC system, such as a location
server and a stub generator, among other things. The SRPC system has been used as a
distributed programming tool in both teaching and research projects for three years.

The SRPC system has another interesting feature. That is, the stub compiler (we call it the stub and driver generator, or SDG in short) not only produces the server and client
stubs, but also creates remote procedures’ framework, makefile, and driver programs for
both server and client. After using make utility, a user can test the program’s executability
by simply executing the two driver programs. This feature will be more attractive when
a programmer is doing prototyping.

3.1 Server Types

The client/server model [13] is used in the SRPC system. An SRPC program has two
parts: a server part and a client part. Usually the server provides a special service or
manages an object. The client requests the service or accesses the object by using the
remote procedures exported by the server.

There are three types of servers in the SRPC system: simple servers, service providing
servers and object managing servers. Figure 1 depicts these three types of servers.

A simple server (Figure 1(a)) is an ordinary server possessing with no fault-tolerant
features. When a simple server fails, all RPCs to it have to be aborted.

A service providing server (Figure 1(b)) has a buddy server running somewhere in the
network (usually on a host different with the server’s), but no communication between
the server and its buddy. When a service providing server fails, an RPC to this server
will be automatically re-directed to its buddy server by the system. As object changes in
the server will not be available in its buddy, a service providing server usually is used in
applications such as pure computation, information retrieval (no update), motor-driven
(no action memory), and so on. It is not suitable to be used to manage any critical object
that might be updated and then shared by clients.

An object managing server (Figure 1(c)) also has a buddy running in the network. It
manages a critical object that might be updated and shared among clients. An RPC to
such a server, if it will change the object state, is actually a nested RPC. That is, when the server receives such a call from a client, it first checks to see whether the call can be executed successfully (e.g. if the necessary write-locks have been obtained or not). If the answer is no, the call is aborted. If the answer is yes, then the server will call its buddy server to perform the operation as well. When the buddy returns successfully, the call commits (the server and its buddy actually perform the call) and the result returns to the client. To ensure the consistency of the objects managed by the server and its buddy, a two-phase commit protocol [6] is used when executing the nested RPC.

Like a service providing server, when an object managing server fails, an RPC to this server will be automatically re-directed to its buddy server by the system.

All buddy servers are simple servers. That means, when a server (service providing or object managing) fails, its buddy server provides alternative service in a simple server manner. Also, when a buddy server fails, a service providing server or an object managing server will be reduced into a simple server.

3.2 The Architecture

The SRPC has the following three components: A Location Server (LS) and its buddy (LS buddy), a system library, and a Stub and Driver Generator (SDG). This section describes the system architecture from a user’s point of view. As server buddies are generally transparent to users, we will omit their descriptions here.

From a programmer’s viewpoint, after the SDG compilation (see Section 5), the server part of an SRPC program is consisted of a server driver, a server stub, and a file which implements all the remote procedures (called procedure file). The server buddies are transparent to users. The server part (or a server program as it is sometimes called) is a “forever” running program which resides on a host and awaits calls from clients. The client part (or a client program) consists of a client driver and a client stub after the SDG compilation. It runs on a host (usually a different host from the server’s host) and makes calls to the server by using the remote procedures exported by the server.

When the client driver makes a call, it goes to the client stub. The client stub then, through the system library, makes use of the client protocol for sending the calling message to the server host. Because the client and the server may use different communication protocols, a client-server protocol converter is used to convert the client’s protocol into server’s protocol. The calling message is then sent to the server. At the server host side, the server’s protocol entity will pass the calling message to the server stub through the system library. The server stub then reports the call to the server driver and an appropriate procedure defined in the procedures file is executed. The result of the call follows the calling route reversely, through the server stub, the server protocol, the system library of the server host, the client-server protocol converter, the system library of the client host, the client stub, back to the client driver. This is called a direct call as the pre-condition of such a call is that the client knows the address of the server before the call.
With the help of the Location Server, the run-time address of a server can be easily accessed. Figure 2 depicts the system architecture using a typical RPC. The dashed line represents the RPCs from the user's viewpoint.

In this project, cross-protocol communication requires an individual converter for each pair of different protocols. It has been noted that this solution is only reasonable for a few protocols. For a large number of protocols, an intermediate protocol description would be better.

3.3 The Location Server

One way of hiding out the implementation details is the use of the Location Server (LS). The LS is used to hide the server locations from users. It maintains a database of server locations and is executed before any other SRPC program is started. After that, it resides on the host and awaits calls from servers and clients.

The Location Server is an object managing server and has a buddy of its own. It has a well-known location, and this location can be easily changed when necessary. The LS itself is implemented by the SRPC system, using the direct calling method.

Usually there should be one LS (called local LS) running on each host for managing locations of that host, and these local LSs report to the "global LS" (like the NCA/RPC's local and global location brokers [14] [9]). In that case the locations of all LSs can also be hidden from users. We have planned to implement this facility.

The following call is used by a server to register itself to the LS:

```c
int registerServer(sn, buddy, imp)
char *sn; /* server name */
char *buddy; /* buddy's name */
struct info *imp; /* implementation info. */
```

where `imp` is a type `struct info` structure and contains many implementation details, such as the server's host name, protocol, and so on. Because the call updates the LS database, it is also directed to the LS buddy. If the call returns OK, the location has been registered and a client can use the following call to find out the location of a server from the LS:

```c
int locateServer(sn, buddy, imp)
char *sn; /* server name */
```
char *buddy; /* server's buddy name */
struct info *imp; /* implementation info. */

If the call returns OK, the location of the server sn is stored in imp and the name of the
server's buddy is stored in buddy for later use. This call does not affect the LS database
state, so there is no hidden LS server and LS buddy communication here. Before a server
is shut down, the following call must be used to un-register the server from the LS:

int unregisterServer(sn)
char *sn; /* server name */

If the call returns OK, the server and its buddy (if any) are all deleted from the LS database.
The system also provides other LS calls for maintaining the LS database.

All the usages of these functions in a server or a client program are automatically
generated by the stub and server generator. A user does not need to look into the details
of these calls if he or she is satisfied with the generated program sections.

3.4 The System Library

The system library is another way of achieving transparency. The library contains all the
low-level and system- and protocol-oriented calls. Its main functions are to make the low-
level facilities transparent to the upper-level programs and make the system as portable
as possible.

The server and client programs must be linked with the system library separately.
Reference [16] contains detailed descriptions of the library calls. All the library calls can
be divided into the following call levels and Figure 3 depicts their relationships:

1. SRPC Level: This is the highest level. It contains calls that deal with RPC-related
   operations.

2. Remote Operation Level: It contains calls that deal with remote operations. These
   remote operations follow the definitions of the OSI Application level primitives [8].

3. Protocol Level: It contains calls that deal with protocol-specific operations.

4. Utility Calls: It contains all the utility calls used in different levels.

4 The Stub And Driver Generator

4.1 Syntax

The purpose of the stub and driver program generator is to generate stubs and driver
programs for server and client programs according to the Server Definition Files (SDF).
Listing 1 is the syntax of a server definition file.

We use a modified BNF to denote the syntax of definition files. The “variable”, “in-
teger”, “string”, “constant”, and “declarator” have the same meanings as in the C pro-
gramming language. Comments are allowed in the definition file. They are defined the
same as in the C programming language (using /* and */).
Listing 1. Server definition file syntax

```
<SDF> ::= BEGIN
    <HEADER>
    [ <CONST> ]
    <FUNCS>
END

<HEADER> ::= Server Name: variable;
            Comment: string;

[Using: <LANGUAGE>;]
Server Protocol: variable;
Client Protocol: variable;
[<BUDDY>]}

<Buddy> ::= Buddy <BDYTYPEx>: variable;
            Using: <LANGUAGE>
            <BDYTYPEx>::= Auto | Forced
            <LANGUAGE>::= C | Pascal
            <CONST>::= constant
            <FUNCS>::= RPC Functions: <RPCS>
            <RPCS>::= <RPC> { <RPC> }
            <RPC>::= Name: string [Update];
            <PARAMS>::=
            <PARAM>::= Param: <CLASx>: declarator;
            <CLASx>::= in | out
```

4.2 Semantics

Most of the descriptions of Listing 1 are self-explanatory. We only highlight the following points:

1. The server’s name is defined as a variable in the C language. This name will be used in many places. For example, it is the key in the LS database to store and access server entities. When the client asks the LS to locate a server, it provides the server’s name defined here. The name is also used as a prefix in naming all the files generated by the SDG. The default language used in the server is the C language.

2. Different protocols can be defined for the server and the client respectively. The buddy, if it is defined, uses the same protocol as the server does. Currently, only three protocols are allowed: Internet_datagram (the UDP protocol), Internet_stream (the TCP protocol), and XNS_datagram (the XNS packet exchange protocol).

3. The <BUDDY> part is optional. If it is not specified, the generated server will be a simple server, otherwise it will be a service providing server or an object managing server, according to some definitions in the <RPCS> part (described below). The <BUDDY> part has a buddy definition and a language definition. The buddy definition defines that whether the buddy’s name and execution is to be determined by the system (Auto) or to be determined by the programmer (Forced). If Auto is defined, the system will generate the buddy server’s name (ServerNameSdy, used for registering and locating it), the buddy’s driver and stub files as well as the makefile, and will treat the following variable as the name of the buddy’s procedure file. Then, the buddy program will be compiled and executed together with the server program. The host of the buddy program will be determined by the system at runtime.

If Forced is defined, the generator will not generate any buddy’s program file and will treat the following variable as the name of the buddy server used for registering and locating. The programming and execution of the buddy server will also be the programmer’s responsibility.

The language definition Using within the BUDDY part defines which language does the buddy program use. The key issue of software fault-tolerant is the design diversity or version independent, and one way of achieving design diversity is through the use of multiple programming languages [12]. Currently only the C programming language is supported in the SRPC system. We have planned to support the Pascal language implementation soon.
4. The <FUNCS> part defines the remote procedures of the server. At least one remote procedure must be defined. Each remote procedure is defined as a name part and a parameter (<PARAMS>) part. The name of a remote procedure is simply a variable, with an optional <UPDATE> definition. The latter definition distinguishes an object managing server with a service providing server. That is, if the <BUDDY> part is defined and the <UPDATE> is defined in any one RPC definition, the server is an object managing server. If only the <BUDDY> part is defined but no <UPDATE> part is defined in any RPC definition, the server is a service providing server. The meaning of the <UPDATE> definition is: if an <UPDATE> is defined following an RPC procedure name, that procedure must be maintained as a nested RPC affecting both the server and the buddy by the server program (See Section 3.1).

There can be zero or several parameters in a procedure, each consisting of a class and a declaration. The class can be in or out, which tells the SRPC system that the parameter is used for input or output, respectively. The declaration part is the same as in the C language. In this version, only simple character string is allowed in parameter definitions. Further extensions are under way.

4.3 Implementation Issues

After a programmer sends a server definition file to the generator, the generator first does syntax checking. If no errors are found, several program source files and a makefile are generated. The subsequent processing is specified by the makefile. That is, when using the make utility, the executable files of both the server and client will be generated. Figure 4 indicates the structure of the processing. The dashed lines represent optional actions.

At least one server definition file must be input to the SDG. If there are more than one server, their SDFs can be input to the SDG simultaneously. If there is only one SDF file, then the generated client driver can execute the server’s procedures one by one. If the buddy part is also specified, the generated client can also call the buddy procedures directly (this is useful in testing the client-buddy communication).

If there are more than one SDF file, then for each server, the SDG will generate one set of server files, one set of client files, and one set of buddy files (if the buddy is defined), respectively. These files are the same as the servers being processed in single file input described above. One additional set of client files, the multi-server client program, will also be generated in this case. The client driver is called a multi-server client driver. It can call all the procedures of all the servers one by one. A further improvement is under way to let the client call these procedures in parallel.

The performance of an RPC in the SRPC system varies, according to which server type is used. Table 1 lists the null RPC performance on a network of HP and SUN workstations, where the server program runs on an HP 715/33 workstation and the server buddy and the client run on two separate SUN 4/75 ELG (33MHZ) workstations. The server (and the buddy, of course) uses the Internet_datagram protocol and the client uses the Internet_stream protocol. We are still investigating the system performance under various circumstances.
<table>
<thead>
<tr>
<th>Server Type</th>
<th>Time</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simple</td>
<td>3.22±0.02ms</td>
</tr>
<tr>
<td>Service-providing</td>
<td>3.37±0.02ms</td>
</tr>
<tr>
<td>Object-managing</td>
<td>5.12±0.04ms</td>
</tr>
</tbody>
</table>

Table 1: Null RPC Performance

5 An Application Example

We use a simple example to show the application of the SRPC system. Suppose we have a server definition file called `sf.def`. It defines a “send-and-forward” system in that the server acts as a message storage and the client acts as both message sender and receiver. Next is the server definition file:

Listing 2. Server definition file example

```c
/* Store and forward: server definition file */

BEGIN
Server Name: sf;
Comment: Store and forward system;
Server Protocol: Internet_datagram;
Client Protocol: Internet_stream;
Buddy Auto: sfBdyUops.c;
      Using: C;

#define MXNAML 64
#define MXMSG 500
#define MXSTR 80

RPC Functions:
Name: storeMsg Update;
Param: in receiver: char receiver[MXNAML];
Param: in msg: char msg[MXMSG];
Param: out stat: char stat[MXSTR];
Name: forwardMsg Update;
Param: in receiver: char receiver[MXNAML];
Param: out msg: char msg[MXMSG];
Name: readMsg;
Param: in receiver: char receiver[MXNAML];
Param: out msg: char msg[MXMSG];
Name: listMsg;

END
```

When this file is input to the generator, the following files will be generated:

- `sf.h` Header file, must be included by server, its buddy and client drivers and stubs.
- `sfSer.c` Server driver file.
- `sfStubSer.c` Server stub file.
- `sfUops.c` Frameworks of server procedures.
- `sfCli.c` Client driver file.
- `sfStubCli.c` Client stub file.
- `sfBdy.c` Server buddy driver file.
- `sfStubBdy.c` Server buddy stub file.
- `makefile` Make file.

After using the make utility (simply use "make" command), three executable files are created:

- `sfSer` Server program.
- `sfCli` Client program.
- `sfBdy` Server buddy program.
Note that the sfops.c file only defines the frameworks of the remote procedures (dummy procedures). Their details are to be programmed by the programmer. The sfBdyOps.c file should be the same as the sfops.c file (the only possible difference happens when the server buddy uses another programming language such as the Pascal, then the affix of the file would be .pas).

The server driver is simple. It does the initialisation first. Then it registers with the LS and invokes the buddy program on a neighbouring host because the buddy is defined as Auto in the SDF file. After that it loops "forever" to process incoming calls until the client issues a "shutdown" call. In that case the server un-registers from the LS and exits. The "un-register" call will automatically un-register the buddy from the LS as well. The incoming calls are handled by the server stub and underlying library functions. Following is the pseudocode listing of the server driver:

Listing 3. Server driver pseudocode
Initialisation (including invoke the buddy);
/* Register the server to the LS */
registerServer("sf", "sfBdy", imp);
while (1) {
    wait for client calls;
    /* comes here only if a client called */
    fork a child process to handle the RPC;
    if the call is "shutdown"
        break;
}
unregisterServer("sf");

The server buddy driver works in the same way as the server program, except that it does not invoke a buddy program. Also the buddy is a simple server and all calls to the buddy will not be nested.

The generated client driver can execute the server’s remote procedures one by one. If the server driver is running and the client driver is invoked, the client driver first lists all the remote procedures provided by the server, and asks the user to choose from the list. The following is the menu displayed for this example:

Available calls:
0  sf$Shutdown
1  sf$storeMsg(receiver, msg, stat)
2  sf$forwardMsg(receiver, msg)
3  sf$readMsg(receiver, msg)
4  sf$listMsg()
Your choice:

After the selection, the input parameters of the named remote procedure are then input from the keyboard. After that, the driver program does some initialisation and the remote procedure is executed and returned results displayed. The actual calling and displaying are handled by the client stub and underlying library functions. The format of all the four RPCs in the client program are the same as the the format listed in the above menu. That is, if the client wants to send a message to a receiver, it does the following call after the receiver's name and the message are input into receiver and msg variables, respectively:

sf$storeMsg(receiver, msg, stat);

Note that the remote procedure’s name is named as a composition of the server’s name sf, a $ sign, and the remote procedure’s name storeMsg in the SDF file. Similarly, if the client wants to receive messages, it does the following call after the receiver’s name receiver is obtained:

114
sf$forwardMsg(receiver, msg);

Before each RPC, a locateServer("en", buddy, imp) call is issued to the LS to return the location of the server and the name of its buddy. The server location is stored in imp and the buddy name is stored in buddy.

The fault-tolerant feature of the system is completely hidden from the user. For this example, all the remote procedure calls from the client program will be first handled by the server. A nested RPC is issued if the incoming call is either sf$storeMsg(receiver, msg, stat) or sf$forwardMsg(receiver, msg). This is because the two RPC functions are marked as Update in the SDF file. The nested RPC will ensure that actions of the incoming call will be made permanent on both the server and its buddy if the call is successful, and no actions of the incoming call will be performed if the call fails. Other two incoming calls, sf$readMsg(receiver, msg) and sf$listMsg(), will be handled by the server only.

If the server fails (that is, the RPC to the server returns an error), the client program will send the RPC to the server’s buddy. The location of the buddy will be determined by another call to the LS:

locateServer(buddy, "", imp)

where buddy is the server buddy’s name obtained during the first call to the LS, and imp stores the location of the buddy.

The cross-protocol communication is also hidden from the user. All the interfaces to the protocol converters (client-LS, client-server, and server-LS) are generated by the SDG (in the stub files) and used automatically by the stubs. If a user only deals with the RPC level, he or she will never notice the underlying protocols used by the server and client programs.

6 Remarks

A system for supporting fault-tolerant, open distributed software development is described in this paper. The system is simple, easy to understand and use, and has the ability of accommodating multiple communication protocols and tolerating server failures. It also has the advantage of producing server and client driver programs and finally executable programs directly from the server definition files. The system has been used as a tool of distributed computing in both third year and graduate level teaching, and has been used by some students in their projects.

In tolerating server failures, similar efforts can be found in the RPC systems that provide replicated server facilities, such as NCA/RPC [14]. But in these systems, the user, instead of the system takes the responsibility of maintaining and programming the functions for object consistency. This is a difficult job for many programmers. Our approach in achieving fault tolerance is similar to the approach used in the ISIS toolkit (of course, ours is more simplified and less powerful). But our system is simple, easy to understand, and easy to use. In our system, we provide a server buddy to tolerate the server’s failure. When the server fails, the client, instead of aborting, can access the server buddy to obtain the alternative service. Also in our system, it is the system, instead of the user, that is responsible of maintaining the consistency of the managed objects.

Providing server and driver programs directly from the server definition file (similar to the interface definition files of other RPC systems) is also an interesting characteristic of our system. It is related to the rapid prototyping of RPC programs [17]. The driver programs are simple, but yet have the advantages of testing the executability of the RPC program immediately after the designing of the SDF file. It is especially useful if the user makes some changes in the SDF file or the procedure file. In that case, these changes will be automatically incorporated into other related program files if the program is re-generated by the stub and driver generator. This will avoid a lot of troubles in the maintenance of consistency of program files.
References


116
Providing Fault-Tolerant Services in an Open Client/Service Paradigm

Wanlei Zhou
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au
Phone: (052)274 240

Abstract
The design and understanding of fault-tolerant open distributed systems is a very difficult task. We have to deal with not only the complex problems of open distributed systems when all the components are well, but also the more complex problems when some of the components fail. This paper proposes a fault-tolerant computing model on open distributed systems. The performance of the model is simulated and analyzed. A prototype implementation of the model using Remote Procedure Calls (RPCs) is also described.

1 Introduction
An open distributed system consists of many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever increasing dependency being placed on open distributed systems, the number of users requiring fault tolerance is likely to increase. Remote Procedure Call (RPC) is perhaps the most popular model used in today's distributed software development and has become a de facto standard for distributed computing. However, two issues are still outstanding in building RPC systems for open distributed systems: the cross-protocol communication between different RPCs and the fault-tolerance features.

This paper is concerned mainly with the software aspects of achieving reliable operations on an open distributed processing environment. Based on a fault-tolerant computing model, a system for supporting fault-tolerant and cross-protocol distributed software development is described. The model and the system design is aimed toward application areas that may involve heterogeneous environments and in which requirements for fault-tolerance are less severe than in, for example, the aerospace field, but in which continuous availability is required in the case of some components failure[5]. The application areas could be, for example, kernel/service pool-based distributed operating systems, supervisory and telecontrol systems, switching systems, process control and data processing. The fault-tolerant technique used is a variation of the recovery blocks technique, and the distributed computing model used is the RPC model.

The reminder of this paper is organised as follows: In Section 2, we summary some notable related work and provide the rationale of our work. In Section 3, we describe the model for supporting open and fault-tolerant services. Section 4 analyses the performance of the model. Section 5 describes the architecture of a prototype implementation of the model for open client/service paradigm. Finally, Section 6 concludes the paper.

2 Related Work and the Rationale
There have been many successful RPC systems since Nelson's work[11], such as Cedar RPC[4], NCA/RPC[13], Sun/RPC[10], HRPC[1], and so on. But little of them consider fault tolerance and cross-protocol communication in their design, or they rely on users to build up these features.

Notable works on incorporating fault tolerance features into RPC systems are the Argus[9], the ISIS[2][3] and an atomic RPC system on ZMOB[8]. The Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. Atomic transactions are serializable and indivisible. A user can also define some atomic objects, such as atomic arrays and atomic record, to provide the additional support needed for atomicity. All the user fault tolerance requirements must be specified in the Argus language.

The ISIS toolkit is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. A special process group, called fault-tolerant process group, is established when a group of processes (servers and clients) are cooperating to perform a distributed computation. Processes in this group can monitor one another and can then take actions based on failures, recoveries, or changes in the status of group members. A collection of reliable multicast protocols are used in ISIS to provide failure atomicity and message ordering.

The atomic RPC system implemented on ZMOB uses sequence numbers and calling paths to control the concurrency and atomicity, and uses checkpointing to maintain the ability of recovering from failures. Users do not have to provide synchronisation and recovery themselves; they only need to specify if atomicity is desired. This frees them from managing much complexity.
But when a server (or a guardian in the Argus) fails to function well, an atomic transaction or an atomic RPC has to be aborted in these systems. This is a violation of our continuous computation requirement. The fault-tolerant process groups of the ISIS can cope with process failures and can maintain continuous computation, but the ISIS toolkit is big and relatively complex to use.

Typical solutions to the cross-protocol communication in RPC systems are the black protocol boxes of the HRPC [11], the special protocol conversion interface [14] and the RPC agent synthesis system [7] for cross-RPC communications.

The HRPC system defines five RPC components: the stub, the binding protocol, the data representation, the transport protocol, and the control protocol. An HRPC client or server and its associated stub can view each of the remaining components as a "black box." These black boxes can be "mixed and matched." The set of protocols to be used is determined at bind time—long after the client and server have been written, the stub has been generated, and the two have been linked.

The special protocol conversion interface proposed in [14] uses an "interface server" to receive a call from the source RPC component (client or server) and to convert it into the call format understood by the destination RPC component (server or client).

The cross-RPC communication agent synthesis system proposed in [7] associates a "client agent" with the client program and a "server agent" with the server program. A "link protocol" is then defined between the two agents and allow them to communicate. The server and the client programs can use different RPC protocols and the associated agents will be responsible of converting these dialect protocols into the link protocol.

But none of the above cross-protocol RPC systems consider fault-tolerance issues. If the server fails, the client simply fails as well.

Incorporating both fault tolerance and cross-protocol communication into RPC systems is clearly an important issue to use RPC's efficiently and reliably in open distributed environments.

In this paper we describe a twin-server model, its performance evaluation, and a prototype system that implements the model for supporting development of fault-tolerant, open distributed software. The prototype implementation incorporates fault tolerance features and protocol converters into the RPC system and makes them transparent to users. The prototype is small, simple, easy to use and also has the advantage of producing server and client driver programs and finally executable programs directly from the server definition files.

3 The Twin-Server Model

The model design is based on the client/service paradigm, which is a higher abstraction of the client/server model [12]. It consists of the following three entities:

- **Server**: A server is an instance of a particular service running on a single machine.
- **Client**: A client is a software entity that exploits services provided by servers. A client can but does not have to interface directly with a human user.

When requesting a service, a client specifies the nature of the service through an attributed name [6]. The exact location of the service and the server that provides such a service will be determined by the system.

We can divide services into two categories: the processing services and the managing services. A processing service provides some processing power to clients. It takes a client's request and processes it. No internal states of the service are changed because of the client's request. We name client requests to these services as processing service calls. A managing service manages one or more objects and clients access these objects by issuing some requests. The states of an object can be changed because of a client's request. We name client requests to these services as managing service calls. A service can be both a processing service and a managing service.

For each service, two servers are maintained to provide the fault-tolerant service. We call them as twins and denote them as $(T_1^1, T_2^1)$. These two servers usually (but not necessarily) locate on different hosts for increasing reliability.

When a client requests a service, the request is mapped to one of the twins. Without loss of generality, suppose that twin server $T_1^1$ is selected. The request is then sent to $T_1^1$ for processing. At the same time, a copy of the request is also sent to $T_2^2$ and is kept in its reserve. If $T_2^2$ has completed the service of the request, it notifies $T_2^2$ and $T_2^2$ will drop the copy of the request from its reserve.

At any time, if one of the twins, say $T_1^1$, dies for some reason, the survivor twin $T_2^2$ will continue the service of $T_1^1$ by using the requests kept in its reserve. Then $T_2^2$ will invoke another server to be its new twin (also named as $T_1^2$). After that, the normal operation of the twins is resumed. Figure 1 depicts the design model.

![Figure 1: The fault-tolerant service model](image-url)
4 Performance Evaluation

Obviously, our model can tolerate single-point failures. This is clearly an advantage compared with the normal single server model. However, every fault-tolerant method employs redundancy, and redundancy may imply increasing costs and/or decreasing performance. This section tries to evaluate the performance of our fault-tolerant model in various circumstances.

4.1 The Queueing Model and Its Simplification

Clearly the two twins are identical and are symmetric. We can then simplify the model into a single service node with a feedback and bulk arrivals. This model is shown in Figure 2.

In Figure 2, $\lambda$ is the arrival rate of the normal requests, $\mu$ is the mean service rate of the server, $p$ describes the probability of the following two actions. The first is that when a request is a managing service call, and therefore needs to be sent to the twin for performing a nested RPC call. The second is that when a twin completes a service to a request, it has to notify the other twin, and therefore the other twin can delete the corresponding request from its reserve. We call it the feedback probability. $X$ is the average size of the bulk arrival, and $q$ is the probability of the bulk arrival (or equivalently the probability of a twin failure).

We are interested in the performance (the response time, for example) changes when the following parameters change: (a) the probability of a twin failure ($q$); (b) the probability of feedback ($p$); and (c) the traffic intensity ($u = \lambda/\mu$). A simulation model is then built and some simulation is performed.

We have the following assumptions when building our simulation:

1. The arrival rate of normal requests $\lambda$ and the mean service rate $\mu$ follow independent exponential distribution.

2. The twin failure probability ($q$) and the feedback probability ($p$) follow a uniform distribution (within $(0, 1)$).

3. The average size of a bulk arrival is the same as the current length of the normal queue.

4. The time used by a twin for moving the requests from its reserve into its normal queue is zero. Also the time used for invoking a new twin is zero.

4.2 Simulation Results

The first result describes the relationship between the system response time and the twin failure probability. Figure 3 depicts the response time when the twin failure probability $q$ changes from 0 to 0.05. The feedback probability $p$ stays at 0.5 and the traffic intensity is $u = 0.33$ (the mean service rate $\mu$ is assumed to be 1 time unit). The conclusion is that, our fault-tolerant model does not impose too much overhead on the system when the twin failure probability is low.

The second simulation result describes the relationship between the response time and the feedback probability. Figure 4 depicts the response time when the feedback probability $p$ changes from 0 to 0.6. The twin failure probability $q$ stays at 0.02 and the traffic intensity is $u = 0.33$.

The conclusion is that, our fault-tolerant computing model will not work properly if most of the requests are managing service calls, and the traffic intensity is relatively high.

The third simulation result relates the response time to the changes of both the twin failure probability and the feedback probability. Figure 5 depicts the response time when the feedback probability $p$ changes from 0 to 0.5, and the twin failure probability $q$ changes from 0 to 0.05. The traffic intensity stays at $u = 0.33$.

Figure 5 confirms our above observations. That is, if both $q$ and $p$ stay low, our model will not impose too much overhead on the system. We can also observe some additional properties. For example, if only $q$ (or $p$, but not both) is high but $p$ (or $q$) is very low, the system overhead will also be low. The system overhead will be drastically increased only when both $q$ and $p$ are high.
The last simulation result shows the relationship between the response time and the traffic intensity. Both $q$ and $p$ are fixed at this time. Figure 6 depicts the response time when the traffic intensity $u$ changes from 0.1 to 0.85. The dotted line shows the response time of the single server model. The middle solid line shows the case when the twin failure probability $q$ stays at $q = 0.01$ (1%) and the feedback probability $p = 0.1$ (10%). The upper solid line shows the case when the twin failure probability $q$ stays at $q = 0.025$ (2.5%) and the feedback probability $p = 0.36$ (36%).

The conclusion is that, if both $q$ and $p$ are relatively low, our model will not impose too much overhead on the system performance. If a service has a less incoming request rate and a higher processing rate, converting the service into our twins model will cause less overhead to the system performance, even if both $q$ and $p$ are relatively high. However, if a service has a very high incoming request rate and its processing rate is low (that means, the service is already congested without introducing the fault-tolerant features), converting the service into our twin model will cause much overhead to the system performance.

5 System Prototype Implementation

A prototype has been built to implement the fault-tolerant service design outlined in Section 3. Here we only present an outline of the architecture of the prototype implementation. Figure 7 depicts the architecture of the prototype system.

The prototype has three main components: the Local SSM (Service-Server Mapping) Agent and the Global SSM Agent, the System Library (including protocol converters), and the Twin Servers.

On each host, there is a local SSM agent. It manages service locations and service-server mappings on its host. Any service running on a host must be registered to the local SSM agent. If that service is to be shared by clients other than the clients of the local host, it has to be registered to the global SSM agent as well. The global SSM agent is responsible of managing service locations and service-server mappings of the whole network. The current prototype does not include the local SSM agents. Therefore, all services are global services and must be registered to the global SSM agent.

In a client/server model, a distributed program consists of a client program part and a server program part. In our system, a client program consists of a Client Driver and a Client Stub. The client driver controls the client's action, and the communication details are hidden in the client stub. The server program consists of a Server Driver, a Server Stub, a Procedures part and a Reserve. Similar to the client program, the server driver controls the server's action, and the communication details are hidden in the server stub. The procedures part realizes the real service operations. The reserve keeps track of the requests sent to its twin.

Because both the local SSM agent and the global SSM agent keep the critical information of services, they must be fault-tolerant themselves. So these SSM agents are also designed to be twins. For a particular SSM agent, the location of its twin is a well-known address.
And for all the local SSM agents, the location of the
global SSM agent is a well known address.

To free programmers from tedious RPC stub pro-
gramming, a rapid prototyping tool [19] is used to pro-
vide a Stub and Driver Generator (SDG) for generating
stub and driver codes according to a Service Definition
Files (SDF).

When the client driver makes a call, it goes to the
client stub. The client stub then, through the system
library, makes use of the client protocol for sending the
calling message to the server host. Because the client
and the server may use different communication proto-
cols, a client-server protocol converter is used to convert
the client's protocol into server's protocol. The calling
message is then sent to the server. At the server host
side, the server's protocol entity will pass the calling
message to the server stub through the system lib-
ary. The server stub then reports the call to the server
driver and an appropriate procedure defined in the pro-
cedures file is executed. The result of the call follows
the calling route reversely, through the server stub, the
server protocol, the system library of the server host,
the client-server protocol converter, the system library
of the client host, the client stub, back to the client
driver.

Before a service can be accessed, the twins have to
register their service to the local and/or global SSM
agent(s). Because the twins and the SSM agents may
use different protocols, a server-SSM agent protocol
converter is provided. Similarly, before a client can access
a service, it has to locate the service through the local
and/or global SSM agent(s). A client-SSM agent proto-
col converter is also provided when it is required. The
twins always use the same protocols, so they can com-
 municate to each other without any protocol conversion.

In this project, cross-protocol communication re-
quires an individual converter for each pair of different
protocols. It has been noted that this solution is only
reasonable for a few protocols. For a large number of
protocols, an intermediate protocol description would be
better.

6 Remarks

We have presented the design, performance eval-
uation and an outline of the prototype implementa-
tion of a fault-tolerant model for an open client/service
paradigm. We feel our model has a wide range of applica-
tions in which continuous availability of services are
required in the case of some components failures. Cur-
rently we are trying to use our model to provide fault-
tolerant features for a microkernel-based distributed oper-
ating system.

References

[1] B. N. Bershad, D. T. Ching, E. D. Lazowska,
J. Satislo, and M. Schwartz. A remote procedure
call facility for interconnecting heterogeneous com-
puter systems. IEEE Transactions on Software En-

munication in the presence of failures. ACM Trans-
1987.

[3] K. P. Birman, A. Schiper, and P. Stephen-
son. Lightweight causal and atomic group multi-
cast. ACM Transactions on Computer Systems,

remote procedure calls. ACM Transactions on

An integrated environment to support construc-
tion of reliable distributed applications (CON-
CORDIA). In Parallel Processing and Applica-
tions, pages 467–473. Elsevier Science Publishers
(North Holland), 1988.

Logical Design. Addison-Wesley Publishing Com-
pany, Reading, Massachusetts, 1991.

agent synthesis system for cross-rpc communica-
tion. IEEE Transactions on Software Engineering,

call. IEEE Transactions on Software Engineering,

[9] B. Liskov. Distributed programing in ARGUS. Com-
munications of the ACM, 31(3):300–312, March

[10] Sun Microsystems. RPC: Remote procedure call
protocol specification version 2 (RFC 1057). In In-
ternet Network Working Group Request for Com-
ments, No. 1057. Network Information Center, SRI

port CSL-81-9, Xerox Palo Alto Research Centre,
May 1981.

computing. In Informatik Fachberichte 95, pages

[13] L. Zahn, T. H. Dieyen, P. J. Leach, E. A. Martin,
N. W. Mishkin, J. N. Pato, and G. L. Wyant. Net-
work Computing Architecture. Prentice-Hall, En-

heterogeneous computer systems. In Proceedings of
the Open Distributed Processing Workshop, Syd-
nex, Australia, January 1990.

[15] W. Zhou. A rapid prototyping system for dis-
tributed information system applications. The Jour-
A Generic Recursive Algorithm for Fault-Tolerant Computing

Wanlei Zhou
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au
Phone: (052) 271 240

Abstract
This paper presents a generic recursive algorithm for fault-tolerant computing. The algorithm uses recursive majority voting to achieve fault tolerance on a multiprocessor system. Both space and time redundancy are employed dynamically in the algorithm. By defining different parameters, the algorithm can be applied to different situations. The correctness and performance analysis of the algorithm are also described.

Key Words: Fault-tolerant computing, multiprocessor systems, performance evaluation.

1 Introduction
As a result of the recent advances in VLSI technology, it is cost-effective now to include a large number of very powerful processors together with their local memories (called computing units, or CUs) in a multiprocessor system. With the increased complexity of computing units as well as the increased complexity of software used for such systems, the number of possible ways in which error conditions can occur in a computing unit may increase [8]. The system will still work if some of its computing units fail. In this case, we have to ensure that we can still obtain correct result even if there exist faulty CUs in the system.

One fundamental consideration in designing and using a multiprocessor system is its reliability, and system reliability clearly becomes more important with the ever increasing dependency being placed on such a system. As the number of CUs in a system grows, its reliability tends to drop rapidly [8], unless a proper fault-tolerant technique is incorporated in the system or a proper fault-tolerant algorithm is used in the computing.

This paper presents a generic recursive algorithm for fault-tolerant computing on a multiprocessor system. The algorithm uses recursive majority voting to achieve fault tolerance on a multiprocessor system. Both space and time redundancy are employed dynamically in the algorithm. By defining different parameters, the algorithm can be applied to different situations.

The paper is arranged as follows. Section 2 presents the generic recursive algorithm. In Section 3, variations of the generic algorithm are described. Section 4 describes the performance evaluation of the algorithm in various circumstances. Section 5 concludes the paper.

2 The Generic Recursive Algorithm

2.1 Assumptions
Suppose we have a CU pool U with N CUs and any CU can be used in our algorithm by specifying its index number in the pool. We do not care here how these CUs are interconnected. User jobs are scheduled to CUs, and results and signatures (required by the voting process or comparators [8]) are obtained from these CUs. A signature may be obtained by using many techniques, such as data
compression [4] from the result. The main requirement for a signature is that it should reflect the different errors in the results.

We allow CUs to have Byzantine failures [7]. That is, a CU of the parallel system can exhibit arbitrary and malicious behaviour perhaps involving collusion with other faulty CUs. We ignore the possible failures of the interconnection network. In fact, as we have assumed Byzantine failures for CUs, failures of the interconnection network can be incorporated into the CU failures.

2.2 Notations and Definitions

We use the following notations to describe the algorithm:

- $P_i$: $i$th CU, $i = 1, 2, \ldots, N$
- $R^T_i$: Result of job $T$ on the $i$th CU
- $S^T_i$: Signature of $R^T_i$ on the $i$th CU
- $LF$: Set of CUs whose signatures do not match with the correct one

We usually omit $T$ if no ambiguity will be caused. The $LF$ set collects the faulty-like CUs with respect to job $T$ and can be used for diagnosis [3].

For the convenience of describing the algorithm, we use the following function to determine the maximum equal signatures from the CU set $V$. Its implementation is feasible.

```c
void MaxEqual(V, t, q, m)
/* where */
V: Input parameter. Set of CUs already used in processing the job;
t: Input parameter. Number of CUs in V;
q: Output parameter. Number of signatures in the maximum equal signature match among CUs in V;
m: Output parameter. The smallest index of the above q CUs.
*/
```

Then the execution of the function $\text{MaxEqual}(V, t, q, m)$ will return $q = 3$ and $m = 2$.

We define a majority number function $r(x)$ as follows, where $x$ is an integer.

$$r(x) = \begin{cases} \frac{x}{2} + 1 & \text{if } x \text{ is even} \\ \frac{x}{2} & \text{if } x \text{ is odd} \end{cases}$$

The function $r(x)$ is to be used later to determine the number of CUs to be scheduled to obtain a majority matching on a job execution.

2.3 The Algorithm

The main goal of the generic recursive algorithm is to use the minimum possible number of CUs to achieve a majority signature match. The generic recursive algorithm works as follows: Let $a$ be an integer. Initially, we schedule $a$ CUs from the CU pool $U$ to execute a task $T$. If the majority of those signatures match with each other, one of the results is output as the correct answer. Otherwise, new CUs from the CU pool $U$ are needed to have a majority matching. If we are now in the stage that $t$ CUs have been used and $q$ is the maximum number of CUs which have the same signatures (that means, $q < r(t)$, the majority number function). By solving the equation of $r(t + k) = q + k$, we know that $k$ more CUs are needed to obtain the minimum set of a majority voting. So, in the next iteration, we schedule $k$ unused CUs to execute the task $T$. The algorithm repeats until a majority matching of signatures is obtained, or there is no more CUs left unused. The algorithm follows:

```
BEGIN
\begin{align*}
t & = k = a; \quad V_1 = \emptyset; \quad V_2 = \{P_{j_1}, \ldots, P_{j_k}\}; \\
LF & = \emptyset; /* initialisation */
\end{align*}
DO WHILE ($t \leq N$)
\begin{align*}
& \quad \text{Schedule job } T \text{ on all } k \text{ CUs of set } V_2; \\
& \quad \text{Obtain } R_{j_n} \text{ and } S_{j_n}, \ n = 1, \ldots, k, \text{ from the execution of } T \text{ on } V_2; \\
& \quad V = V_1 \cup V_2; \quad t = |V|; \\
& \quad \text{MaxEqual}(V, t, q, m); /* determine the maximum equal signatures from } V */ \\
& \quad \text{IF } (q = r(t)) \text{ THEN /* voting successful}
\begin{align*}
& \quad \text{Output } R_m \text{ as the correct result;} \\
& \quad \text{Add } \{P_j \mid (S_j \neq S_m) \text{AND}(P_j \in V)\} \text{ to } LF; /* \text{the faulty-like CUs */}
\end{align*}
\end{align*}
END
```

For example, let $V = \{P_1, P_2, \ldots, P_7\}$ and (of course) $t = 7$. If the matching signatures can be grouped as:

$$S_1 = S_4; \quad S_2 = S_5 = S_6; \quad S_3; \quad S_7$$
Exit "Job Completed";

ELSIF /* voting failed; prepare next iteration */

V_i = V; t = |V_i|;

k = (t - 2 \times q + 1);

V_2 = \{P_{s_n} \mid (n = 1, \cdots, k) \AND (P_{s_n} \in U) \AND (P_{s_n} \notin V_1)\};

END IF;

END DO;

/* next is the case of degraded result output */

k = N - t; /* use all remaining CUs */

V_2 = \{P_{s_n} \mid (n = 1, \cdots, k) \AND (P_{s_n} \in U) \AND (P_{s_n} \notin V_1)\};

Schedule job T on all k CUs set V_2;

Obtain R_{s_n} and S_{s_n}, n = 1, \cdots, k, from the execution of T on V_2;

V = V_1 \cup V_2; t = |V|;

MaxEqual(V, t, q, m); /* determine the maximum equal signatures from V */

Output (P_{m}, q) as the degraded result;

Add \{P_j \mid (S_j \neq S_m) \AND (P_j \in V)\} to LF; /* the faulty-like CUs */

Exit "Job Completed. Degraded result.";

END

2.4 Correctness Of The Algorithm

We only outline the informal proofs for the correctness of the algorithm.

Assertion 1: The algorithm terminates after a finite time of computation.

Proof: There are three termination possibilities:

- Term 1: When the signatures of the first a CUs match with, one of their results is output to the user and the algorithm terminates;
- Term 2: When majority signatures match in the used CU set V has been achieved; and
- Term 3: When there are not enough unused CUs left, and all remaining CUs are used in the last computation and the algorithm terminates after the degraded result is reported.

Term 1 is obvious. Because of the initialisation in lines 2 and 3, the set V of line 7 contains only the initial a CUs. If the majority voting has been reached during the first execution attempt, the algorithm will terminate at line 12.

Because all used CUs are put into set V_1 and the newly needed CUs for job T are taken from those unused (relative to job T) CUs (that is, each CU is only used once during a job's computation), and there are finite N CUs in the CU pool U, so if majority signatures match before all CUs are exhausted, termination possibility Term 2 will be reached at line 12. If no majority signature matches occur, the CU pool will be exhausted eventually and the last termination possibility Term 3 will be reached at line 27.

Assertion 2: The algorithm produces either a majority signature match result or a degraded result if it terminates.

Proof: We denote by V the used CU set for job T, and by S the maximum number of equal signatures of all CUs in V. It is evident that at Term 1, \[ |V| = a \text{ and } S = q \], where \( q = a \) or \( q = r(n) \). So, we have a majority match. At Term 2, \[ |V| = k + t \], where \( k \) is the number of newly assigned CUs and \( t \) is the number of CUs that have been used before, and \( S = q = r(k + t) \). So, we still have a majority match. At Term 3, \[ |V| = N \], the number of CUs in the whole CU pool, and the degraded result is output.

3 Variations of the Algorithm

One variation of the algorithm is to set \( a = 2 \) and to limit \( k = 1 \). The majority voting part is also reduced to an agreement of two signatures. That is, at first, two CUs are scheduled and the signatures compared. The correct result is output when both CUs have the same signature. Otherwise, one more CU is used in the next iteration to execute
the task and the signature of it is compared with all the existing signatures. The correct result is output as long as there is an agreement of two signatures among all the used CUs and the next iteration always uses one new CU. This is an algorithm proposed by Agrawal [1] [2] and improved by Zhou [9]. We name this algorithm as Algorithm A.

More variations can be achieved by setting up different initial values of \(\alpha\). In that case, the above generic algorithm will behave differently. For example, let \(\alpha = 2\) (it is the minimum value allowed). The algorithm produces a correct result during the first iteration as long as these two CUs produce the same signature. Otherwise, one more CU is needed to produce a majority voting in the second iteration \((k = 1)\). The number of CUs used in the following iterations changes according to the number of maximum matching signatures. For example, if we have used the CUs of \(\{P_1, P_2, \ldots, P_t\}\) (that is, at line 7 of the algorithm, we have \(t = 7\)), and there's no majority matching yet. The following possibilities exist:

- None of the signatures match with each other. That is, \(S_1 \neq S_2 \neq \cdots \neq S_t\). From line 8 of the algorithm, we know \(q = 1\). From line 15 we know that the new CUs need in the next iteration to achieve a majority signature match is \(k = (t - 2 \times q + 1) = 6\).
- The maximum number of matching signatures is 2. In that case, we have \(q = 2\) at line 8 and \(k = 4\) at line 15. So, 4 more new CUs are needed in the next iteration.
- The maximum number of matching signatures is 3. That is, \(q = 3\) and \(k = 2\). We need 2 more new CUs in the next iteration.

We name this algorithm as Algorithm B.

If we let \(\alpha = 3\), then the algorithm produces a correct result during the first iteration as long as two out of these three CUs produce the same signature. The algorithm goes into the second iteration only if all of these three CUs produce different signatures. In that case, two more CUs are needed to produce a majority voting in the second iteration. The number of CUs used in further iteration changes according to the number of maximum matching signatures. We name this algorithm as Algorithm C.

The obvious disadvantage of Algorithm A is the lack of majority voting except the first iteration. It, however, uses less space (CUs) redundancy than other algorithms do (say, for example, Algorithms B and C).

Clearly, Algorithm C will have a larger probability of success rate than that of Algorithm B (and A, of course) during the first iteration if other conditions remain the same. More initial CUs can be deployed in order to achieve a higher success rate during the first iteration. The ultimate number of CUs used in the first iteration is of course \(N\), the number of CUs in the whole CU pool \(U\). This reduces the algorithm into a simple majority voting algorithm. The obvious advantage of using more CUs in the first iteration is to save time (less time redundancy). The obvious disadvantage of this, however, is to have higher space (device) redundancy.

### 4 Performance Simulation and Analysis

In this section we use simulation to evaluate the performance of the algorithm under various circumstances. At first we describe our assumptions used in the simulation. Then we present the performance metrics and simulation results. Finally, the simulation results are analysed.

#### 4.1 Descriptions and Assumptions

Our simulation tool is based on MacDougall's snmp [6]. Some modifications are made in order to suit the non-queueing system characteristics of the algorithms. In order to make the simulation models more realistic, jobs entering the system are divided into two classes. Class 0 is the ordinary jobs, i.e., jobs that do not require fault tolerant computation; Class 1 is the fault tolerant jobs, and they may use algorithm A, B, or C, respectively. We have the following descriptions and assumptions during the simulation:

- Suppose there are \(N\) identical CUs in the CU pool, and they are numbered from 1 to \(N\). For any CU \(P_j, j = 1, \ldots, N\), its service discipline is FCFS. All jobs in the same class have the same service time distribution at \(P_j\), and the
service time distribution is negative exponential. We use $\mu_i$ to denote the service rate for class $i$ ($i = 0, 1$) jobs. We also assume that service rates are independent of the system status.

- There are two arrival streams to the system, one for each job class. We assume that all streams are Poisson streams. The arrival rate of class $i$ jobs is $\lambda_i$ ($i = 0, 1$).

- Let $T$ be a class 1 job. A failure mode of a CU with respect to a job $T$ means that the result and signature produced by the CU when executing $T$ are incorrect. Let $p$ be the probability of a CU being good with respect to $T$, and $E$ the set of failure modes of a CU with respect to $T$, $|E| = M$. That is, if $P_j$, $j = 1, \ldots, N$ fails with respect to $T$, then the failure mode $e_i$, $i = 1, \ldots, M$ can be one of the $M$ elements in $E$. We also assume that $e_i$ is equally distributed in $E$. Of course, here we assume that the signature technique can differ all these failure modes.

4.2 Performance Metrics

We use $R$ to represent the algorithm used for fault-tolerance ($R = A, B, C$). The performance metrics we have used in our simulation are:

- Average response time: denoted as $ART_i$, where $i = 0, 1$. It denotes the average response time of class $i$ jobs.

- Average Number of error results: denoted as $AER$. This is the error results occurred during the simulation period. For algorithm $A$, an error result will occur if two CUs produce the same failure modes; while for algorithm $B$ and $C$, an error result will occur if the majority signature match reached but the result is of a failure mode.

- Average CUs used: denoted as $APU$. This is the average number of CUs used by an algorithm in order to obtain a result. It denotes the resource needs of the algorithm.

The above notations will be denoted as $ART_i^R$, $AER^R$, or $APU^R$, respectively, when we want to emphasize a particular algorithm $R$ is used.

![Figure 1](image1.png)  
Figure 1: Average response time (ART) changes when $N$ changes ($m = 500$ and $p = 0.90$). (a). Class 0 jobs; (b). Class 1 jobs.

4.3 Results Analysis

The following simulation results are achieved when $\mu_0 = 5.0$, $\mu_1 = 10.0$, $\lambda_0 = 4.0$, and $\lambda_1 = 12.0$. Similar results will be obtained if slightly different $\mu_i$ and $\lambda_i$ are used.

Figure 1 is the average response time (ART) variations when the number of CUs in the pool $U$ changes. We know from the simulation that if $N$ is large enough, class 0 jobs will not be affected because of the different algorithms used in class 1 jobs. Also class 1 jobs will not be affected by class 0 jobs if $N$ is large enough. This is also true from intuition. Usually a multiprocessor system is comprised of a lot of CUs, and therefore our algorithm is applicable to most of the multiprocessor systems.
One of the important performance metrics for a fault tolerant algorithm is the average number of error results (AER) occurred during the computation. It may be affected by the number of fault modes \( m \), or by the probability of a CU being good with respect to a job \( p \). Figure 2 gives the AER comparison of the three algorithms.

Figure 2(a) is the AER variation when \( m \) changes and \( p=0.90 \) (notice that \( m \) is expressed by logarithm of 10). We can see that AER decreases when \( m \) increases. When \( m > 1000 \), the three algorithms produce the same performance. But when \( m \) is small, algorithm B and C are much better than algorithm A. Also when \( m < 100 \) (\( \lg(m) < 2 \) in the diagram), \( AER^C \) increases very slowly, while \( AER^B \) and \( AER^A \) increase sharply. The conclusion is that the algorithm B and C are suitable when \( m \) is in median size, while algorithm C is very suitable when \( m \) is small.

Figure 2(b) is the AER variations when \( p \) changes and \( m=500 \). We can see that if \( p > 0.95 \), the three algorithms produce the same performance. But when \( p \) decreases, \( AER^A \) increases sharply. While the AER's of the other two algorithms increase very little. The conclusion is that, the algorithm A is suitable to use when the probability of a CU being good with respect to a job \( p \) is high, while the algorithm B and C are suitable when \( p \) is low.

Figure 3 shows the changes of the resource need
and response time when $p$ changes. Figure 3(a) is the average number of CUs used (APU) when $p$ changes. Here we set $m = 500$. It can be seen that when $p > 0.85$, $APU^A = APU^B$, and $APU^C \approx APU^B + 1$. This is because that at that situation, algorithm $A$ and $B$ will output almost all correct results at the first iteration of computation, and they all use two CUs at the first step. Algorithm $C$ will also output most correct results at the first step, but it uses three CUs in that step. So we have the above relationships.

When $p$ decreases, $APU^A$ increases a little, but $APU^B$ and $APU^C$ increase rapidly. Also $APU^B$ and $APU^C$ keep a difference of about one CU. This is true because algorithm $C$ uses one more CU at the first step, and the two algorithms use the same majority voting strategy after the failure of the first step. The conclusion is that, when $p$ is small, the algorithms $B$ and $C$ will use more resources than the algorithm $A$. There is little differences in resource needs, however, when $p$ is large.

Figure 3(b) is the average response time (ART) variations when $p$ changes. Here we also set $m = 500$. It can be seen that when $p > 0.85$, $ART^A = ART^B$, and $ART^C$ is greater than $ART^B$ a little. When $p$ decreases, the response time of all algorithms increase and $ART^B$ and $ART^C$ increase faster than $ART^A$. The conclusion is that, if $p$ is small, the algorithm $B$ and $C$ will take much longer to obtain a result than that of the algorithm $A$. But when $p$ is large, there is little differences in response times among these algorithms.

5 Remarks

A generic recursive algorithm for fault-tolerant computing is described and its correctness and performance discussed. The algorithm can be set to various forms. Three most important forms of the generic algorithm are identified (they are named as algorithm $A$, $B$, and $C$, respectively) and a simulation is carried out to evaluate the performance of these special forms of the algorithm. The results show that in general, when $p$ (the probability of a CU being good with respect to a job) is small, or $m$ (the number of failure modes of a CU with respect to a job) is small, the algorithms $B$ and $C$ have good performance. However, when both $p$ and $m$ are large, the algorithm $A$ is better.

References


A Tool for Layered Analysing and Debugging of Distributed Programs

Wanlei Zhou

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
wanlei@deakin.edu.au

Abstract

Analysing and debugging distributed programs is much more difficult than analysing and debugging sequential programs. One of the reasons is the communication among programs (processes) which may happen concurrently and nondeterministically. To be able to analyse such communication events is therefore an essential task for any distributed program analyser/debugger. This paper describes the design and implementation of a tool for layered analysing and debugging of distributed programs. In the highest level, the tool displays the communication relationships between programs (eg, server and client programs). In the second level, the communication between processes is displayed. In the third level, the communication between events is displayed. The fourth level is the lowest level: it uses a text editor to show the relevant statements that carry out the communication. The tool has been regularly used in analysing and debugging student assignments on distributed programming for three years.

1 Introduction

A distributed program can be viewed as a group of program parts (PPs, each PP can be a process or even a program. An example is the client and server program parts of a distributed program) that work together on a single task, and the concurrency and communication among these parts are the main reasons that make debugging of distributed programs difficult [4, 14]. To be able to analyse such concurrent and communicating events is therefore an essential task for any distributed program debugger/analyser.

This paper describes the design and implementation of a tool for layered analysing and debugging of distributed programs. The tool has been implemented on networks consisting of DEC/HP/SUN workstations and it has been regularly used in teaching of courses related to distributed systems since 1993 (mainly used in analysing and debugging student assignments on distributed programming). The tool helps a user to analyse and debug a distributed program in a top-down fashion. In the top level, the tool displays the communication relationships between program parts (eg, server and client programs). The top level gives a user the overall function of all program parts involved in the distributed computing. In the second level, the communication between processes is displayed. Because a program part can split into several processes and these processes can run concurrently with processes from other program parts, the second level therefore gives a user a clear picture of co-operations among all (or a selected group of) processes. In the third level, events related to some particular communication and a partial ordering among these events are displayed. The forth level is the lowest level: it displays the relevant statements that carry out some particular communication. The first three levels help the localisation of bugs and the forth level helps the analysis/fixing of bugs.

The remainder of this paper is organised as follows: Section 2 presents an example to illustrate our layered analysing and debugging of distributed programs. Section 3 describes the design issues of the tool. Section 4 describes issues related to the tool’s implementation. Section 5 presents some related work. Finally, Section 6 summaries the paper.

2 An Example

We use a simple debugging example to illustrate the layered analysing and debugging process. When teaching “Distributed Computing” course, a student asked me to find out why his exercise program did not work properly. The program was a “Send-and-forward” system. The server (named as Server) acted like a message storage. A user used the client program (named as Client) to send a message (with a receiver’s name) to the server. The server kept the message until the addressed receiver (also a client program) asked the server to forward messages.

We started the server part of the student program first. Then a client part was used to send a message to the server. The normal message exchange should be as follows:

1. The client sends a request to the server.

131
2. The server processes the request and sends an acknowledgement back to the client.

3. Upon receiving the acknowledgement, the client sends the message to the server.

4. The server sends an acknowledgement back to the client after receiving the message.

When the student’s program was executed, both the server and client program parts were hung up. It was difficult to guess what was happening inside these two program parts. We then used our tool to record the events of the program and obtained the top-level diagram of Figure 1.

![Figure 1: Debugging: The Level 1 Picture](image)

From this diagram we know that the client sent the request, the server acknowledged and then the client sent the message. We further used the second level to locate the processes that were responsible for the failed communication. Figure 2 is the process level diagram.

Three processes were involved. For better understanding of the communication we asked the tool to display event tables for all these three processes. Table 1, Table 2 and Table 3 are these tables (where E represents Event Symbol). The second column of these tables lists the event symbols. The correspondence of these event symbols and their detailed event names (see Section 3.1) is given in Table 4.

From the event tables we knew that process 6917 (client process) sent a request to the server process 6911. Then process 6911 forked a child process (6922) to manage the communication with the client. Process 6922 then acknowledged to process 6917. After that process 6917 sent the message to process 6922. It seemed that process 6917 worked normal. What we needed to know more was the communication details between process 6917 and process 6922. So the third level diagram of Figure 3 was used to show the event relations of these two processes.

![Figure 2: Debugging: The Level 2 Picture](image)

### Table 1: Event Table of Process 6911

<table>
<thead>
<tr>
<th>No.</th>
<th>E</th>
<th>Meaning</th>
<th>Pred</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a1</td>
<td>Server begins</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>a2</td>
<td>Create socket</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>a3</td>
<td>Bind to a name</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>a4</td>
<td>Receive request</td>
<td>6917.1</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>a5</td>
<td>Fork a child process</td>
<td>-</td>
<td>6922.1</td>
</tr>
<tr>
<td>6</td>
<td>a6</td>
<td>Forced exit</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

From the level 3 diagram, we knew that the possible error locations were:

1. The program section between events 6 and 7 of process 6922 (the final acknowledgement was not sent properly).

2. The program section between events 6 and 7 of process 6917 (the acknowledgement was not received properly).

We analysed the program section of possibility 1 using a text editor and found out that inside the message storing function, some value of the client socket address was mistakenly re-assigned. That caused the sendto() call of the server child process to send the acknowledgement to an unknown address. After fixed the mistake, the program worked correctly.

### Table 2: Event Table of Process 6917

<table>
<thead>
<tr>
<th>No.</th>
<th>E</th>
<th>Meaning</th>
<th>Pred</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>b1</td>
<td>Client begins</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>b2</td>
<td>Create socket</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>b3</td>
<td>Bind to a name</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>b4</td>
<td>Send request</td>
<td>-</td>
<td>6911.4</td>
</tr>
<tr>
<td>5</td>
<td>b5</td>
<td>Receive ACK</td>
<td>6922.5</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>b6</td>
<td>Send MSG</td>
<td>-</td>
<td>6922.6</td>
</tr>
<tr>
<td>7</td>
<td>b7</td>
<td>Forced exit</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>
Table 3: Event Table of Process 6922

<table>
<thead>
<tr>
<th>Event symbol</th>
<th>Event name</th>
</tr>
</thead>
<tbody>
<tr>
<td>c1</td>
<td>57121282b3c1.00.0282c-225150000000</td>
</tr>
<tr>
<td>c2</td>
<td>57121282b3c7.01.0282c-225150000000</td>
</tr>
<tr>
<td>a1</td>
<td>57121282b3d3.02.0282c-226150000000</td>
</tr>
<tr>
<td>a2</td>
<td>57121282b3f1.03.0282c-226150000000</td>
</tr>
<tr>
<td>c3</td>
<td>57121282b3f1.04.0282c-226150000000</td>
</tr>
<tr>
<td>c4</td>
<td>57121282b3f3.05.0282c-226150000000</td>
</tr>
<tr>
<td>b1</td>
<td>57121282b3f4.00.0282c-226150000000</td>
</tr>
<tr>
<td>b2</td>
<td>57121282b3f4.01.0282c-226150000000</td>
</tr>
<tr>
<td>b3</td>
<td>57121282b3f4.02.0282c-226150000000</td>
</tr>
<tr>
<td>b4</td>
<td>57121282b3f4.03.0282c-226150000000</td>
</tr>
<tr>
<td>b5</td>
<td>57121282b3f4.04.0282c-226150000000</td>
</tr>
<tr>
<td>a3</td>
<td>57121282b3f4.05.0282c-226150000000</td>
</tr>
<tr>
<td>a4</td>
<td>57121282b3f4.06.0282c-226150000000</td>
</tr>
<tr>
<td>a5</td>
<td>57121282b3f4.07.0282c-226150000000</td>
</tr>
<tr>
<td>a6</td>
<td>57121282b3f4.08.0282c-226150000000</td>
</tr>
<tr>
<td>a7</td>
<td>57121282b3f4.09.0282c-226150000000</td>
</tr>
<tr>
<td>c5</td>
<td>57121282b3f4.10.0282c-226150000000</td>
</tr>
</tbody>
</table>

Table 4: Event symbols and event names

3 Design Issues

3.1 Definitions

Before describing the architecture of the tool, we present several definitions which are important in our discussion.

In order to monitor events we have to assign each event a name. Definition 1 is the method which can uniquely name any event.

Definition 1. The name of an event (event name) is a "unique ID" (UID), defined as a string of characters and an optional user attached suffix. The format is t.r.h[.suffix] where t is the timestamp, which is stamped by the local host; r is a sequential number; h is the host ID; and suffix is an optional suffix name (a character string) which is defined by a user.

If e is an event name, we use e.t to denote the timestamp, e.n to denote the sequential number, e.h to denote the host, and e.a to denote the suffix of event name e, respectively. En.h can be used to identify which host the event happened, and en.t denotes the occurrence time (relative to the local host) of the event. In the case that n events happened simultaneously at the same host, en.r is used to differentiate these n events. It is not difficult to insure that no n (n is a hexadecimal number and n ≤ 255) adjacent sequential numbers generated are the same. For example, we can have a sequential number generator (which is accessed sequentially by the event name generator) that generates numbers from 0 to 255 each time it is accessed. If the number reaches 255, it goes back to 0 and the circle begins again. So all events of a distributed program can be uniquely identified by using their event names if the maximum number of concurrent events in any host is n ≤ 255. The assignment and usage of the suffix will be described in Section 4.4.

Definition 2. A preliminary event e is defined as a pair (f, m). Where f is called a fact and m a message. A fact is a thing which happens during a program's execution. Notice that not all facts of preliminary events are interesting to a programmer, but they may happen during the program's execution. A fact can be, for example, the creation and destruction of a process, the issuing of a message sending call, or the issuing of a message receiving call, and so on. A message is the information attached to the fact, such as the parameter values of a message sending call.

The basic relation between preliminary events is the happened before relation introduced by Lamport [7]. This relation can be easily extended to cover process creation and termination as well as message-passing events [6]. Definition 3 defines the relationship between preliminary events in our system.

Definition 3. Let E = {e_i} be the set of all events of a distributed program. If
event \( e_1 \) causes the occurrence of event \( e_2 \), or \( e_2 \) immediately follows the occurrence of \( e_1 \) within the same process, we say that \( e_1 \) is a predecessor of \( e_2 \) and \( e_2 \) a successor of \( e_1 \). Especially, if \( e_1 \) and \( e_2 \) are in different processes, we call \( e_1 \) a remote predecessor of \( e_2 \) and \( e_2 \) a remote successor of \( e_1 \). This is denoted as \( e_1 \preceq e_2 \). If \( e_1 \) is a predecessor of \( e_2 \) and \( e_2 \) a predecessor of \( e_3 \), then we say \( e_1 \) is also a predecessor of \( e_3 \).

For example, if \( e_1 \) is the event “issuing a request sending call” of a client program, and \( e_2 \) is the event “receiving a remote request” of the server program. If \( e_2 \) happens because of \( e_1 \)’s happening, then \( e_1 \) is a predecessor (remote predecessor) of \( e_2 \) and \( e_2 \) a successor (remote successor) of \( e_1 \). If \( e_3 \) is the event “receiving acknowledgement” of the same client and happens immediately after \( e_1 \), then \( e_1 \) is a predecessor of \( e_3 \) and \( e_2 \) is a remote predecessor of \( e_3 \). Also, \( e_3 \) is a successor of \( e_1 \) and a remote successor of \( e_2 \). It is easy to know that \((E, \preceq)\) is a partially ordered set.

Sometimes a user may be interested in the combination of several events. For example, if a server has two remote procedures that will access an object, it is interesting to see if these two procedures are all called during the execution, or to know the execution order of them. We give the following definitions.

**Definition 4.** Let \( e_1, e_2 : E \) and \( e_1 = (f_1, m_1), e_2 = (f_2, m_2) \). By \( f_1 \ast f_2 \) we mean that the happening of \( e_2 \) follows the happening of \( e_1 \), that is, \( e_1 \preceq e_2 \) holds. By \( f_1 + f_2 \) we mean that we cannot tell which of \( e_1 \) and \( e_2 \) happened first, that is, there is no predecessor relation exists between these two events. We can view \( \ast \) as sequential and \( + \) as concurrent. By \( f_1 \cap f_2 \) we mean that both \( e_1 \) and \( e_2 \) occur. By \( f_1 \cup f_2 \) we mean that either or both \( e_1 \) and \( e_2 \) occur (if \( e_1 \) does not occur, we denote that as \(-e_1\)). So we have

\[
f_1 \cap f_2 = \begin{cases} f_1 \ast f_2 & \text{if } e_1 \preceq e_2 \\ f_2 \ast f_1 & \text{if } e_2 \preceq e_1 \\ f_1 + f_2 & \text{otherwise} \end{cases}
\]

and \( f_1 \cup f_2 = f_1 \cap f_2 \) or \( f_1 \) or \( f_2 \). Similarly, by \( m_1 \ast m_2 \) we mean that message \( m_1 \) is followed by \( m_2 \) and by \( m_1 + m_2 \) we mean that two messages are independent each other.

**Definition 5.** If \( e_1 = (f_1, m_1) \) and \( e_2 = (f_2, m_2) \) are events, then

\[
e_1 \cap e_2 = (f_1 \cap f_2, m_a),
\]

\[
e_1 \cup e_2 = (f_1 \cup f_2, m_b),
\]

\[
e_1 \ast e_2 = (f_1 \ast f_2, m_1 \ast m_2),
\]

\[
e_1 + e_2 = (f_1 + f_2, m_1 + m_2).
\]

We call these new events combined events. Combined events are usually defined by programmers.

The priority of the above operators is, from high to low, \( \ast \), \( \cup \), \( + \) and \( \cap \). So the expression \( e_1 \ast e_2 \cup e_3 \) is actually \( e_1 \ast ((e_2 \cup e_3) \cup e_4) \) + e_5. It can be proved that if \( E \) is the set of all (preliminary and combined) events of a distributed program, then \((E, \preceq)\) is still a partial order set.

### 3.2 The Structure

The tool consists of a controller and a group of managing servers. The controller has two main parts: a user interface (including a command interpreter and I/O) and a filter, while a managing server (MS) consists of a server and an event database. Each host which has program parts being analysed/debugged on it has a managing server. The controller can be invoked at any host. By communicating with each related MS, the controller can present the monitored results to the user. Of course, several controllers can be invoked by several users simultaneously. Figure 4 illustrates the structure of the tool.

Three steps are taken during analysing and debugging. At the monitoring step, all events that happened on one host are monitored by the local MS and

---

*Figure 4: The Monitor Structure*
recorded into the local event database. All preliminary events including communication-oriented calls, process forks and exits in both client and server program parts are monitored, and a user can also define some combined events through the Event Definition File (EDF) and let the monitor to record them. After all events are recorded, the program uses the ordering step to order events. At that time, each MS exchanges remote predecessor / successor information through the controller and has all remote relationship ordered. Then, local predecessor / successor relationship is established by each MS over its local event database respectively. The last step is debugging. By combining the results on all related event databases, the filter can present the execution trace of the distributed program in several levels.

4 Implementation Issues

4.1 Debugging Library

No doubt an effective distributed debugger has to be deeply embedded into the operating system or even has the help of dedicated hardware components for achieving sufficient speed and transparency. Because of the difficulty of modifying operating systems and obtaining hardware support, most debugger and monitor researchers use software techniques as a substitute. This makes the implementation much easier, but the performance is not very good, especially for real time systems (may be even completely not suitable for real time programs). At this stage, we provided a debugging library which has to be linked with a program being debugged. This library provides replacements for the following BSD4.3 UNIX operating system calls:

- fork(): create a child process
- signal(): signal a process
- kill(): signal a process
- socket(): create a socket
- bind(): bind a name to a socket
- listen(): listen for a connection
- accept(): accept a connection request
- connect(): initiate a connect
- close(): close a socket
- write(): send out a message
- read(): receive a message
- send(): send out a message
- recv(): receive a message
- sendto(): send out a message
- recvfrom(): receive a message
- sendmsg(): send out a message
- recvmsg(): receive a message

These replacements first perform some work required by the tool, such as reporting to the local MS of the event's happening, and then do the normal work of the original calls. After the programmer thinks the program has been debugged, the program can be re-linked with ordinary libraries.

A utility program is used to change all the above system calls of a program into their corresponding debugging library calls (the replacements). This is done by converting the name of a call into a string of capital letters. For example, a fork() call is converted into a FORK() call, which is a debugging library call. After the pre-processing, the program can be compiled normally and linked with the debugging library. Another utility program is used to change all the debugging library calls back to normal system calls after the debugging.

4.2 Managing Server

On each host, there is a managing server (MS) which consists of a server and an event database. Each entry of the event database stores an event and includes the following fields:

- name: event name. A UID
- pid: process ID
- pgm.name: program part name
- p.name: predecessor event names. A UID array
- s.name: successor event names. A UID array
- fact.info: fact information
- message: message part of the event

The fact information of an event is a character string that provides readable information of the fact. For preliminary events, if the user does not provide fact information, such information will be assigned by the debugging library. For example, they may be "begin sendto() call" or "fork new process", etc. Otherwise the user provided character string is used. For combined events, they are assigned by programmers.

The message part of an event is stored as a string of bytes and type information. The filter uses the type information to illustrate the byte string and displays the result to the user through the command interpreter.

An MS has the following functions:

1. Database management. Responsible for the management of the local event database. The database is protected by the MS and any access of it must go through the MS.

2. Event logging. When an event occurs, it is logged by the local MS into the event database. Because the events may happen very fast (or even concurrently), while the logging of an event requires some amount of time, we use an event queue to queue up all events waiting to be inserted into the database. All events are queued into a event queue after their happening, and the local MS looks at the queue and puts the events into the event database.

3. Communicating with the controller. All communications among MSs are conducted by the controller. A lot of commands are issued by the controller and performed by MSs. This is the only way a user can access the event database.
Storing all events of a distributed program execution may cost too much memory, especially for large programs. Facilities are provided to allow a user to select and store events that may be interesting. Also a user can let the monitor store only the fact parts of some events and whole information of other events.

4.3 Controller

The controller consists of a command interpreter and a filter. The command interpreter accepts and analyses commands from a user and controls the filter to perform the required functions.

The filter has three main functions. Firstly, it maintains the communication between the command interpreter and MSs. After the command interpreter accepted and interpreted a command, it is passed to the filter to have the appropriate MSs to execute the command. Then the results of the execution are interpreted and passed to the command interpreter through the filter. secondly, the filter maintains the communication between the MSs. In that case the programming of an MS is much simpler. The last function of the filter is to interpret the message part of an event. When the user requests to view the message part during debugging step, the filter will find the appropriate message and use the type information to illustrate the byte string, and then pass the result to the command interpreter.

4.4 Event Definition File

All preliminary events are automatically logged by the tool if the program being analyzed/debugged is linked with the debugging library. Sometimes a user may find it is more convenient to define some new events during debugging. The Event definition file (EDF) is used for that purpose. The following is a very simple EDF which defines a combined event BothCreated as the intersection of two preliminary events CreateSock1 and CreateSock2. That is, if both preliminary events happened, then the combined event also happened.

BEGIN
preliminary Event:
    CreatedSock1, CreatedSock2;

Combined Event:
    (BothCreated, "Both sockets are created");
    BothCreated = CreatedSock1 \& CreatedSock2;
END

Several steps are needed to use an EDF. At first, the user inserts into the program parts being analyzed/debugged an affix definition function before each preliminary event which is to be used in the EDF. The format of the affix definition function (defined in the debugging library) is affix.define(affix), where affix can be any character string. This string will be appended to the preliminary event name when the event happens. Secondly, the combined events are built by using these affix names, and the EDF is read and evaluated by the controller. When any of these affixed preliminary events happens, they are sent to the controller by the local MSs (of course, the local MSs also record them as usual). The controller then evaluates the combined events expressions and records them into the event database if any of them is true. The predecessors of a combined event are all the events (preliminary and/or combined events) in the right hand side of the event expression.

4.5 Ordering Events

As we have known, the time system of each host in an LAN is not synchronised, so we can not use the timestamp in definition 1 to order all events. But the timestamp can certainly be used to order events happened in one process because they always remain within the same host. That is, events within each process of a distributed program can be fully ordered. But it is impossible to fully order events of different processes. As mentioned earlier, there do exist some partial ordering relationship among these communication and process fork events. The following steps are used to establish the partial ordering among all events:

1. Communication related predecessors. When a communication related event happens (for example, the issuing of a sendto() or recvfrom() call), it will cause the happening of an event which belongs to another process (and also possibly, on another host). In that case, the first event is changed (by the debugging library) to carry not only the original information, but also the event's name. On the other hand, the second event is also changed (by the debugging library) to not only receive the original information, but also the first event's name, and this name is stored by the local MS as the predecessor of the second event.

2. Process fork related predecessors. When a process fork event happens, it will cause a new process to be setup and executed. This event is changed (by the debugging library) to carry the name of the event, and the first event of the new process will use the carried name as its predecessor event.

3. Combined event related predecessors. When an event with an affix definition happens, it will cause the controller to evaluate the related combined event expressions. So, the name of the event is sent to the controller and stored as one of the predecessors of the related combined events.

4. Form all remote successors. In (1), (2), and (3), all remote predecessors will be established after the termination (normal or forced) of the programs being debugged. The remote successors are built by each involved MS and the controller at this moment. Each MS checks all events in its local database. If its event e has a remote predecessor named f, then the MS will be responsible for storing e as f's successor. It is easy if e and f are in the same database (for example, the fork
events). Otherwise, \( f.h \) is used to locate the 
MS it belongs to and \( f \)'s successor will be stored by 
the communication of these two MSs through the 
filter.

5. Form all other successors and predecessors. All 
the events within a process are ordered by their 
timestamps and their predecessors / successors 
are stored by using this order. In one process, 
the predecessor of event \( e \) is event \( d \) if \( d.t \) is im-
mEDIATELY LESS THAN \( e.t \). And the successor of \( e \) 
is event \( f \) if \( f.t \) is IMMEDIATELY GREATER THAN \( e.t \). 
This ordering is performed by each MS concur-
rently.

By using the above method, all communication 
events can be partially ordered by predecessor / suc-
cessor relations. For the events within a single pro-
cess, we can fully order them by their timestamps. 
Combining these two relations together, we can have 
some partial ordering over all events of a distributed 
program.

4.6 Layered Debugging

After the monitoring and ordering steps, the user 
goes into the third step, the debugging. As we men-
tioned before, the first thing of debugging is to locate 
the bugs. A top-down view of the program is a suit-
able way of localising bugs.

At the top level (program level), communication 
between program parts is displayed. A distributed 
program may have many program parts. The tool 
provides a facility to let the user select program parts 
for display. Because in each event entry, there is a 
field \( pga.name \) specifies the name of the program part 
(Section 4.2), we use this information and the event 
ordering to draw the communication diagram between 
program parts. At this level, no event detail is dis-
played. The user can have a nice top-view of the pro-
gram communication.

From the top level, the user may have some idea 
that which part of the program probably has a bug. 
So some relevant processes can be selected and the 
second level (process level) will display the commu-
ication between these selected processes. If nothing 
can be found at the top level, the user can ask the tool 
to display all the processes in the second level. We 
also use a field \( psegid \) (see Section 4.2) in every event 
entry and the event ordering to draw the diagram.

Usually from the second level, some processes can 
be selected for further investigation. The third level 
(detailed level) displays the events and communica-
tions between selected processes (of course, it can also 
separate all the events of the program). In 
this level, the event numbers are used and the user 
can consult these numbers with the event tables (see 
below).

From the third level, the bug locations will be 
found and the relevant program segments will be dis-
played using a text editor in the forth level (text level).

The user can analyse the program segment and fix the 
bug here.

During the displays, the user can view event de-
tails in two levels. At the table level, the events of a 
selected process is displayed in a table form (event 
table). Then the user can select an event within an 
event table and ask the tool to display its details (de-
tail level).

5 Related Work

A lot of techniques have been derived for program-
ers to improve the debugging process. There are 
two major approaches: debugging with repeated exe-
cution of the program (or cyclic debugging) and 
debugging with trace of program execution [9]. In cyclic 
debugging, a user executes the program in a con-
trolled manner until an error is detected. The pro-
gram can be re-executed to produce the same exe-
cution behaviour. This is a very convenient way for 
small programs or programs that have little commu-
nication among their concurrent parts. But for a 
larger distributed program, executing the entire com-
putation several times while repeatedly setting break-
points may be very costly. Also, sometimes the re-
execution of a distributed program may not result in 
the same behaviour because of the nondeterministic 
characteristics. In debugging with trace, no repro-
ducibility behaviour is needed. The generated trace is 
no longer nondeterministic and can be analysed in 
any controlled ways that a programmer prefers. But 
the generating of the trace may be very costly in time 
and space, and also the events in the trace are still not 
fully ordered. Some techniques are needed to analyse 
the program trace. In this paper, the latter method 
is used.

Event-Based distributed debugging has been 
widely discussed. Bates and Widenen [2] used a 
method called behavioural abstraction to hierarchi-
cally define higher level events in terms of sequences 
of primitive events (such as process creation, page fault, 
and message exchanges). In his latter work [1], Bates 
described a system (EDBA) for debugging on hetero-
genous distributed systems based on his behaviour 
abstraction.

In a distributed system, sometimes the re-
execution of a long program is very costly. Replay is a 
technique that allows a user to examine the course of an 
erroneous execution without re-executing the pro-
gram [12]. LeBlanc and Robbins [8] provided some 
degree of replay in their debugger. After the collec-
tion of all events, the events are displayed sequen-
tially. Both single step and continuous display are 
supported.

Debugging a distributed program can be divided 
into two phases. At the first phase, called localisation, 
we need to locate which part of the program has a bug. 
Then at the second phase, called analysis/fixing, we 
analyse the code that may cause the bug and fix it. 
Unfortunately almost all existing distributed debug-
gers only provide communication events occurred in lower level. They assist in fixing a bug after having obtained a rough idea about the bug’s localisation. This only provides information for the second debugging phase. To dig out the possible bug locations using the existing debuggers is a difficult job as there are usually many events involved.

A project of high-level debugger for parallel programs is described by Caerts et al [3]. They use several abstraction levels (from the coarse-grain interacting processes or threads to textual representation of the program) in their debugger. A top-down method following the abstraction levels is used to locate a possible bug. But the debugger is limited to message-passing on shared memory systems.

There have been many efforts in visualising distributed and parallel program execution to facilitate the understanding of these programs [13] [11]. However, most of these tools are too complex to use and therefore are not useful for novices. These existing tools usually do not allow easy study and experimentation with programs. Some existing tools provide graphical pictures of algorithms or data structures in action [10] [6], but do not display the structures and source code of programs.

6 Summary

The design and a preliminary implementation of a tool for analysing and debugging distributed programs is described in this paper. The tool has several managing servers which record the events of program parts of their hosts into their local event databases. By using an ordering scheme, all events of a distributed program can be partially ordered, and the event graphs in different levels and the relevant event tables can be built. These event graphs and tables are then used to locate the possible bug positions in a top-down manner. Facilities are also provided to define combined events and to view the details of the events. The tool has been regularly used in analysing and debugging student assignments on distributed programming since 1993.

References


A Tool for Assisting the Understanding of Distributed Programs

Wanlei Zhou
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract
Understanding distributed programs is much more difficult than understanding sequential programs. One of the reasons is the communication among programs (processes) which may happen concurrently and nondeterministically. To be able to analyze such communication events is therefore an essential task for understanding any distributed programs. This paper describes a tool for assisting the understanding of distributed programs. The tool analyses a distributed program in four levels. In the highest level, the tool displays the communication relationships between programs (e.g., server and client programs). In the second level, the communication between processes is displayed. In the third level, the communication between events is displayed. The forth level is the lowest level; it uses a text editor to show the relevant statements that carry out the communication. The tool has been used in the teaching of courses related to distributed computing since 1993.

1 Introduction
Distributed computing has become the norm of today's computing. However, the understanding of distributed programs is much more difficult than the understanding of sequential programs [2].

A distributed program can be viewed as a group of program parts (each program part can be a process or even a program). An example is the client and server program parts of a distributed program that work together on a single task. The concurrency and communication among these program parts are the main reasons that make understanding of distributed programs difficult [1]. To be able to analyze such concurrent and communicating events is therefore an essential task for understanding any distributed programs.

There have been many efforts in visualising distributed and parallel program execution to facilitate the understanding of these programs [6] [5]. However, most of these tools are too complex to use and therefore are not useful for novices. These existing tools usually do not allow easy study and experimentation with programs. Some existing tools provide graphical pictures of algorithms or data structures in action [4] [3], but do not display the structures and source code of programs.

This paper describes a tool for assisting the understanding of distributed programs. It is a significant improvement of our earlier distributed monitor [8] [9] and has been used in teaching of courses related to distributed systems since 1993. The tool helps a user to understand a distributed program in a top-down fashion. In the top level, the tool displays the communication relationships between program parts (e.g., server and client programs). The top level gives a user the overall function of all program parts involved in the distributed computing. In the second level, the communication between processes is displayed. Because a program part can split into several pro-
cesses and these processes can run concurrently with processes from other program parts, the second level therefore gives a user a clear picture of co-operations among all (or a selected group of) processes. In the third level, events related to some particular communication and a partial ordering among these events are displayed. The fourth level is the lowest level: it displays the relevant statements that carry out some particular communication.

The rest of the paper is organised as follows. Section 2 presents an example to illustrate our top-down fashioned understanding of a distributed program. Section 3 briefly describes the structure of the tool. Section 4 describes issues related to the tool’s implementation. Section 5 concludes the paper.

2 An Example

We use a simple example to illustrate the understanding process of a distributed program using our tool. The program is a “send-and-forward” system. The server program (named as ServerPrg) acts like a message storage. A user used the client program (named as ClientPrg) to send a message (with a receiver’s name) to the server. The server keeps the message until the addressed receiver uses the client program to ask the server to forward messages.

![Figure 1: The Level 1 Picture](image)

We start the server part of the program first. Then a client part is used to send a message to the server. After the program is executed under the monitoring of the tool, we have the top-level diagram of Figure 1.

From this diagram we know that the client initialises a request. The server acknowledges and then the client sends the message to the server. The server then sends back an acknowledgement to the client after receiving the message. We further use the second level to find out the processes that are involved in the communication. Figure 2 is the process level diagram.

![Figure 2: The Level 2 Picture](image)

Three processes are involved. Process 6911 is the main process of the server program and process 6911 is the process of the client program. When the server receives a request from the client, it splits into two processes. The parent process (6911) waits for requests from other clients and the child process (6922) is then responsible for all the communications between the server and the client.

We can further examine the detailed communication events between process 6917 and process 6922. The third level diagram of Figure 3 is used to show the event relations of these two processes. It can be seen from Figure 3 that, after splitting from the parent process, the child process of the server program sends an acknowledgement to the client process. The client process sends the message to the child process.
Figure 3: The Level 3 Picture

Then finally the child process returns an acknowledgement to the client process. These communication events are clearly numbered in Figure 3.

<table>
<thead>
<tr>
<th>No.</th>
<th>E</th>
<th>Meaning</th>
<th>Pred</th>
<th>Succ</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>a₁</td>
<td>Server begins</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>a₂</td>
<td>Create a socket</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>a₃</td>
<td>Bind to a name</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>a₄</td>
<td>Wait for requests</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>a₅</td>
<td>Receive request</td>
<td>6917</td>
<td>-</td>
</tr>
<tr>
<td>6</td>
<td>a₆</td>
<td>Fork a child process</td>
<td>6922</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>a₇</td>
<td>Wait for requests</td>
<td>-</td>
<td>-</td>
</tr>
</tbody>
</table>

Table 1: Event Table of Process 6911 (E: Event Symbol)

For better understanding of the communication we ask the tool to display event tables for all these three processes. Table 1, Table 2 and Table 3 are these tables. The third column of these tables lists the event symbols. The correspondence of these event symbols and their detailed event names (see Section 3.1) is given in Table 4.

Combining the level 3 diagram and the event tables can facilitate the understanding of the events associated with the processes. For example, from the event tables we know that process 6917 (client process) sends a request to the server process 6911 in event b₄. Then process 6911 forks a child process (6922) in event a₆ to manage the communication with the client. Process 6922 then acknowledges to process 6917 in event c₅. After that process 6917 sends the message to process 6922 in event b₅ and process 6922 sends back an acknowledgement to process 6917 in event c₇ after process 6922 receives the message.

If we want to view the detailed programming code on communication events, we can use the fourth level diagram to invoke a text editor to examine the code.

3 Structure of the Tool

3.1 Definitions

Before describing the structure of the tool, we present several definitions which are important in our discussion.

In order to monitor events we have to assign each event a name. Definition 1 is the method which can uniquely name any event.

Definition 1. The name of an event (event
Table 4: Event symbols and event names

<table>
<thead>
<tr>
<th>Event symbol</th>
<th>Event name</th>
</tr>
</thead>
<tbody>
<tr>
<td>a₁</td>
<td>602fae1f3ef.00.0282c22615000000</td>
</tr>
<tr>
<td>a₂</td>
<td>602fae135e1.01.0282c22615000000</td>
</tr>
<tr>
<td>a₃</td>
<td>602fae16e4.02.0282c226150000000</td>
</tr>
<tr>
<td>a₄</td>
<td>602fae17e9a.03.0282c226150000000</td>
</tr>
<tr>
<td>a₅</td>
<td>602fae56d5a1.04.0282c226150000000</td>
</tr>
<tr>
<td>a₆</td>
<td>602fae56c793f.05.0282c226150000000</td>
</tr>
<tr>
<td>a₇</td>
<td>602fae56c6a3.11.0282c226150000000</td>
</tr>
<tr>
<td>b₁</td>
<td>602fae55f976e.00.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₂</td>
<td>602fae55cb98.01.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₃</td>
<td>602fae55ce07.02.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₄</td>
<td>602fae55d5a2.03.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₅</td>
<td>602fae56fe21.04.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₆</td>
<td>602fae56f919.05.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₇</td>
<td>602fae56f614.06.0282c2e0d20000000</td>
</tr>
<tr>
<td>b₈</td>
<td>602fae55a919.07.0282c2e0d20000000</td>
</tr>
<tr>
<td>c₁</td>
<td>602fae55e293a.08.0282c226150000000</td>
</tr>
<tr>
<td>c₂</td>
<td>602fae55e2169.09.0282c226150000000</td>
</tr>
<tr>
<td>c₃</td>
<td>602fae55e293a.09.0282c226150000000</td>
</tr>
<tr>
<td>c₄</td>
<td>602fae55e2169.09.0282c226150000000</td>
</tr>
<tr>
<td>c₅</td>
<td>602fae55e2169.09.0282c226150000000</td>
</tr>
<tr>
<td>c₆</td>
<td>602fae55e2169.09.0282c226150000000</td>
</tr>
<tr>
<td>c₇</td>
<td>602fae56e39e.01.0282c226150000000</td>
</tr>
<tr>
<td>c₈</td>
<td>602fae56e71.01.0282c226150000000</td>
</tr>
</tbody>
</table>

Definition 2. An event e is defined as a pair (f, m). Where f is called a fact and m a message. A fact is a thing which happens during a program's execution. Notice that not all facts of events are interesting to a programmer, but they may happen during the program's execution. A fact can be, for example, the creation and destruction of a program, the issuing of a message sending call, or the issuing of a message receiving call, and so on. A message is the information attached to the fact, such as the parameter values of a message sending call.

Definition 3. Let E = {e₁} be the set of all events of a distributed program. If event e₁ causes the occurrence of event e₂, or e₂ immediately follows the occurrence of e₁ within the same process, we say that e₁ is a predecessor of e₂, and e₂ a successor of e₁. Especially, if e₁ and e₂ are in different processes, we call e₁ a remote predecessor of e₂ and e₂ a remote successor of e₁. This is denoted as e₁ ≤ e₂. If e₁ is a predecessor of e₂ and e₂ is a predecessor of e₃, then we say e₁ is also a predecessor of e₃.

For example, if e₁ is the event “issuing a request sending call” of a client program, and e₂ is the event “receiving a remote request” of the server program. If e₂ happens because of e₁’s happening, then e₁ is a predecessor (remote predecessor) of e₂ and e₂ is a successor (remote successor) of e₁. If e₃ is the event...
oriented calls process forks and exits in both client and server program parts are monitored. At the ordering step, the partial order among events are established. At this step, each MS exchanges remote predecessor / successor information through the controller and has all remote relationship ordered. Then, local predecessor / successor relationship is established by each MS over its local event database respectively. The last step is presentation. By combining the results on all related event databases, the filter can present the execution trace of the distributed program in several levels.

4 Implementation Issues

4.1 Monitoring Library

We provided a debugging library which has to be linked with a program being monitored. This library provides replacements for the following BSD4.3 UNIX operating system calls:

- fork(): create a child process
- signal(): signal a process
- kill(): signal a process
- socket(): create a socket
- bind(): bind a name to a socket
- listen(): listen for a connection
- accept(): accept a connection request
- connect(): initiate a connection
- close(): close a socket
- write(): send out a message
- read(): receive a message
- send(): send out a message
- recv(): receive a message
- sendto(): send out a message
- recvfrom(): receive a message
- sendmsg(): send out a message
- recvmsg(): receive a message

These replacements first perform some work required by the monitor, such as reporting to the local MS of the event's happening, and then do the normal work of the original calls.

4.2 Managing Server

On each host, there is a managing server (MS) which consists of a server and an event database. Each entry of the event database stores an event and includes the following fields:
name  event name. A UID
pid   process ID
pgm.name program part name
p.name  predecessor event names. A UID array
s.name  successor event names. A UID array
fact.info  fact information
message  message part of the event

The fact information of an event is a character string that provides readable information of the fact. If the user does not provide fact information, such information will be assigned by the monitoring library. For example, they may be "begin sendto() call" or "fork new process", etc. Otherwise the user provided character string is used.

An MS has the following functions:

1. Database management. Responsible for the management of the local event database. The database is protected by the MS and any access of it must go through the MS.

2. Event logging. When an event occurs, it is logged by the local MS into the event database. Because events may happen very fast (or even concurrently), while the logging of an event requires some amount of time, we use an event queue to queue up all events waiting to be inserted into the database. All events are queued into an event queue after their happening, and the local MS looks at the queue and puts the events into the event database.

3. Communicating with the controller. All communications among MSs are conducted by the controller. Many commands can be issued by the controller and performed by MSs. This is the only way a user can access the event database.

Storing all events of a distributed program execution may cost too much memory, especially for large programs. Facilities are provided to allow a user to select and store events that may be interesting. Also a user can let the tool store only the fact parts of some events and whole information of other events.

4.3 Ordering Events

As we have known, the time system of each host in an LAN is not synchronized, so we can not use the timestamp in definition 1 to order all events. But the timestamp can certainly be used to order events happened in one process because they always remain within the same host. That is, events within each process of a distributed program can be fully ordered. But it is impossible to fully order events of different processes. As mentioned earlier, there do exist some partial ordering relationship among these communication and process fork events. The following steps are used to establish the partial ordering among all events:

1. Communication related predecessors. When a communication related event happens (for example, the issuing of a sendto() or recvfrom() call), it will cause the happening of an event which belongs to another process (and also possibly, on another host). In that case, the first event is changed (by the debugging library) to carry not only the original information, but also the event's name. On the other hand, the second event is also changed (by the debugging library) to not only receive the original information, but also the first event's name, and this name is stored by the local MS as the predecessor of the second event.

2. Process fork related predecessors. When a process fork event happens, it will cause a new process to be setup and executed. This event is changed (by the debugging library) to carry the name of the event, and the first event of the new process will use the carried name as its predecessor event.

3. Form all remote successors. Steps (1) and (2) will establish all remote predecessors. The remote successors are built by each involved MS and the controller. Each MS checks all events in its local database. If its event e has a remote predecessor named f, then the MS will be responsible for storing e as f's successor. It is easy if e and f are in the same database (for example, the fork events). Otherwise f.a is used to locate the
MS it belongs to and f's successor will be stored by the communication of these two MSs through the filter.

4. Form all other successors and predecessors. All the events within a process are ordered by their timestamps and their predecessors / successors are stored by using this order. In one process, the predecessor of event e is event d if d.t is immediately less than e.t. And the successor of e is event f if f.t is immediately greater than e.t. This ordering is performed by each MS concurrently.

By using the above method, all communication events can be partially ordered by predecessor / successor relation. For the events within a single process, we can fully order them by their timestamps. Combining these two relations together, we can have some partial ordering over all events of a distributed program.

4.4 Layered Presentation

After the monitoring and ordering steps, the user goes into the third step, presentation. A suitable way of understanding a distributed program is to use the top-down method where the structures of program parts are presented first, then the structure of processes, then events, and finally the contents of the programming code.

At the top level (program level), communication between program parts is displayed. A distributed program may have many program parts. The tool provides a facility to let the user select program parts for display. Because in each event entry, there is a field pgnamename specifies the name of the program part (Section 4.2), we use this information and the event ordering to draw the communication diagram between program parts. At this level, no event detail is displayed. The user can have a nice top-view of the program communication.

From the top level, the user may have some idea that which part of the program is of interesting. So some (or all) relevant processes can be selected and the second level (process level) will display the communication between these selected processes. We also use a field (pid, see Section 4.2) in every event entry and the event ordering to draw the diagram.

Usually from the second level, some processes can be selected for further investigation. The third level (detailed level) displays the events and communications between selected processes (of course, it can also display all the events relations of the program). In this level, the event numbers are used and the user can consult these numbers with the event tables.

From the third level, the user may identify the events that are essential for the understanding of the distributed program. The fourth level will then use a text editor to display the relevant program segments. The user can study the program segments to understanding the details of the communication.

During the displays, the user can view event details in two levels. At the table level, the events of a selected process is displayed in a table form (event table). Then the user can select an event within an event table and ask the tool to display its details (detail level, it displays the message part of the event).

4.5 Evaluation

An evaluation of the tool was conducted in late 1993. Two groups of third-year students (each group has three students) were given the source code and executables of a distributed calendar system [7] and were asked to produce a report on the structure of the system within a week. The system has two server programs, two client programs, and some library routines. It has about 3,000 lines of code. The first group was given our tool while the second was not. After two days, the first group presented their report while the second group was only able to present their report after 5 days. The main problem that the second group faced was that they did not know where to start. In contrast, the first group used our tool to analyze the distributed program systematically in four levels.

5 Summary

The design and implementation of a tool for assisting the understanding of distributed programs is described in this paper. The tool has several managing servers which record the events of program parts of their hosts into their local event databases. By using an ordering scheme, all events of a distributed program can be partially ordered, and the event graphs
in different levels and the relevant event tables can be built. These event graphs and tables are then used to present the program to the user in a top-down manner.

References


Performance Evaluation of Nested Transactions on Locally Distributed Database Systems

Wanlei Zhou
School of Computing and Mathematics
Deakin University, Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract
This paper describes an execution time estimating model for nested transactions running on locally distributed database systems (LDDBS). At first the model of nested transactions and the model of an LDDBS are established. The performance evaluation model of nested transactions is then built and some calculation and simulation results of the model are analysed.

1 Introduction
The demand for high transaction processing rate has motivated the development of multiprocessor or locally distributed database systems (LDDBS) [2]. An LDDBS has a tightly interconnection among its front-end system and the database servers, and therefore the communication delay is negligible compared to the database transaction delays [3]. Figure 1 depicts the architecture of an LDDBS.

![Architecture of an LDDBS](image)

Previous studies on the performance of nested transactions on LDDBSs (e.g., [3], [1], and [4]) have two limitations. Firstly they concentrate on performance evaluation of some specific routing strategies, not general routing strategies. Secondly, they all use queuing theory to evaluate their models and therefore the result reflects the performance of a class of transactions and the system, not a specific transaction.

Our work is to study the performance of nested transactions when (1) general routing strategies, instead of specific routing strategies, are used; (2) the user's viewpoint, instead of the system's viewpoint, is used in modeling nested transaction performance; and (3) routing of tasks to database servers (processors) as well as routing of tasks to shared data items are considered.

2 The Nested Transaction Model
Definition 1. A flat transaction is a unit of database processing that cannot be divided and it can be allocated independently to a database server for processing. We use $A_{flat}$ as the data transaction.

Definition 2. A nested transaction is constructed through the application of the following abstract syntax:

$$
\begin{align*}
\text{NestedTrans} &::= S_{block} | \text{P_block} \\
&\quad | L_{block} | \text{Flat_Tran}
\end{align*}
$$

$$
\begin{align*}
S_{block} &::= \text{BEGIN} \{ \text{NestedTrans} \} \text{END} \\
P_{block} &::= \text{IBEGIN} \{ \text{NestedTrans} \} \text{ICEND} \\
L_{block} &::= \text{SEBEGIN} \{ \text{NestedTrans} \} \text{SEEND} \\
\text{Flat_Tran} &::= \text{Flat transaction}
\end{align*}
$$

Equivalently we can associate each nested transaction with a flowgraph. The nodes of the graph represent the starting and synchronisation points of flat transactions and block constructors, and the arcs of the graph represent flat transactions of the nested transaction. Two special nodes, denoted by $s$ and $f$, indicate the starting point and finish point of the nested transaction, respectively.

Assertion 1. The flowgraph of a nested transaction is acyclic.

3 The Evaluation Model
3.1 The Locally Distributed Database System
The LDDBS system studied in this paper has a similar architecture as the system described in [4]. Database servers are denoted by $S_i$ where $i = 1, 2, \ldots, n$ and the databases managed by these database servers are denoted by $DB_i$ where $i = 1, 2, \ldots, n$, respectively.

A routing strategy can be represented as a function $u : A \rightarrow S$, where $A$ is the set of flat transactions of a nested transaction and $S$ is the set of database servers.
$A = \{A_1, A_2, \cdots, A_m\}$ and $S = \{S_1, S_2, \cdots, S_n\}$. Further, when a flat transaction $A_i$ is routed to a database server, we use another function $\nu: A \to D$ to represent the mapping of $A_i$'s to particular shared data items. Here $D$ is the set of shared data items in a particular database server ($D = \{D_1, D_2, \cdots, D_l\}$).

Let us model the required execution time of a flat transaction by a function $t: A \to \text{real numbers}$. Given a flat transaction $A_i \in A$, the front-end system tells us that $A_i$ is going to be executed on the database server $\nu(A_i)$, that $A_i$ is going to use a shared data item $\omega(A_i)$, and that the execution of $A_i$ will require time $t(A_i)$.

We use a function $f(k)$ to represent the slow-down factor when $k$ flat transactions are routed to one database server at the same time and denote the execution time of each $A_i$ by $t(A_i) \times f(k)$. It is reasonable to assume that $f(k) = 1$ when $k = 1$ and $1 \leq f(k) \leq k$ when $k > 1$.

### 3.2 The Nondeterministic Algorithm

We use the following nondeterministic algorithm to evaluate the execution time of a nested transaction, where $\alpha$, $\beta$, and $\gamma$ represent "allocated", "executing", and "completed", respectively.

Assign Completion and CurrentTime to 0;
Mark all arcs $A_i$ leaving start node $\alpha(u(A_i), v(A_i));$
while there exist a arc which can execute {
    select an $\alpha$ arc $A_i$ which can execute;
    mark it $\beta$;
    $A_i$.CompletionTime = CurrentTime + $t(A_i) \times f(k)$;
} while there exist arc marked $\beta$
select a $\beta$ arc $A_i$ with minimum CompletionTime;
mark it $\gamma$;
CurrentTime = $A_i$.CompleteTime;
denote target node of $A_i$ by $y$;
if the arc just marked belongs to a selective block {
    mark all arcs of that selective block $\gamma$;
    set all execution time of these arcs to CurrentTime;
} if all arcs $A_i$ entering $y$ are marked $\gamma$
    $y$.CompletionTime = max($A_i$.CompletionTime); mark $\alpha(u(A_i), v(A_i))$ to all arcs leaving $y$;
} while there exist $\alpha$ arc which can execute {
    select an $\alpha$ arc $A_i$ which can execute;
    mark it $\beta$;
    $A_i$.CompletionTime = CurrentTime + $t(A_i) \times f(k)$;
} The nested transaction execution time = the CompletionTime of the finish node;

### 3.3 Performance Measures

Given a routing strategy $u$ and $v$, a nested transaction $g$ will execute (according to our model) in a time denoted $T_{uv}(g)$. This time depends on the topology of the flowgraph of $g$, and the routing of the $n$ flat transactions over database servers and shared data items. Our problem of interest is that repeated executions may take place, with a different routing strategy map each time. If we denote by $U$ and $V$ the sets of routing maps of interest, then the standard metrics are

\[
T_{\text{min}}(g) = \min_{u \in U, v \in V} T_{uv}(g),
\]

\[
T_{\text{max}}(g) = \max_{u \in U, v \in V} T_{uv}(g),
\]

\[
T_{\text{avg}}(g) = \frac{1}{|U||V|} \sum_{u \in U, v \in V} pr(u, v)
\]

where $pr$ is a probability function defined over the sets of maps $U$ and $V$. We are only interested in the equal-probable case, that is, all routing strategies are treated equally. So we have,

\[
pr(u, v) = \begin{cases} \frac{1}{|E|} & \text{for } U, \text{ where } E = \text{card}(U) \\ \frac{1}{|F|} & \text{for } V, \text{ where } F = \text{card}(V) \end{cases}
\]

### 4 Simple Case Solutions

It is easy to estimate the execution time of a single block by simply adding the execution times of all its flat transactions. That is, let $S$ be a serial block defined as $S(A_1, A_2, \cdots, A_m)$, where $A_i$'s ($i = 1, 2, \cdots, m$) are the flat transactions. Let $T(A_i)$ ($i = 1, 2, \cdots, m$) be the estimated execution times of those transactions, then the execution time of $S$ is $T_S = \sum_{i=1}^{m} T(A_i)$. But for general situations, it is very difficult to obtain the explicit solution.

We characterise the sets $U$ and $V$ of routing strategy maps as follows. Each $u \in U$ is a map from the finite set $C$ to the finite set $S$ and each $v \in V$ is a map from the finite set $C$ to the finite set $D$ where we denote

\[
n = \text{card}(S), m = \text{card}(C), \text{ and } l = \text{card}(D).
\]

**Assertion 2.** If $g$ is a simple parallel block, then

\[
T_{\text{max}}(g) = m
\]

\[
T_{\text{min}}(g) = \left\lceil \frac{m}{l} \right\rceil
\]

**Assertion 3.** If $z$ flat transactions are routed to a database server $S_r$, then the average execution time of all these $z$ flat transactions is:

\[
T_{\text{avg}}(z) = \frac{1}{E} \sum_{y=1}^{z} y \times Q_y^{z/l}
\]

354

148
where $E$ and $Q_{x}^{l}$ are defined as follows:

$$E = d_{x}^{(2)}(l, [0, \infty)) = \left( \begin{array}{c} l + x - 1 \\ l - 1 \end{array} \right)$$

$$Q_{x}^{l} = d_{x}^{(2)}(l, [0, y]) - d_{x}^{(2)}(l, [0, y - 1])$$

$y = 1, 2, \ldots, z$ and $d_{x}^{(2)}(l, [0, 0]) = 0$.

$$d_{x}^{(2)}(l, [0, y]) = \sum_{0 \leq s \leq l} (-1)^{s} \binom{l}{s} \binom{l + z - (y - 1)s - 1}{l - 1}.$$ 

Assertion 4. If $g$ is a simple parallel block, then the average execution time of $g$ is

$$T_{avg}(g) = \frac{1}{F} \sum_{x=1}^{z} x \times Q_{x}^{m/n} \times T_{avg}(x).$$

where

$$F = d_{m}^{(2)}(n, [0, \infty)) = \left( \begin{array}{c} n + m - 1 \\ n - 1 \end{array} \right).$$

5 Lower and Upper Bounds Estimation

We define an extended parallel block as

COBEGIN

$E_{1}; E_{2}; \ldots; E_{k}$

COEND

where $E_{i}$ is a simple parallel block or an extended parallel block. This is a recursive definition.

Assertion 5. If $g$ is a nested transaction built by using serial and parallel block constructors, then

$$T_{min}(g) \leq TL_{avg}(g) \leq T_{avg}(g).$$

where

$$TL_{avg}(A) = MAX_{i=1}^{k} (TL_{avg}(E_{i})).$$

If $E_{i}$ is a simple parallel block, then $TL_{avg}(E_{i})$ is calculated by using the equations in section 4. That is, in this case we have $TL_{avg}(E_{i}) = T_{avg}(E_{i})$. Otherwise, $TL_{avg}(E_{i})$ is calculated by using the definition again. So the calculation is also recursive.

If $g$ is a nested transaction, we define the level of a flat transaction $A_{i} \in g$ as the length of the longest path from the start node $s$ of the nested transaction flowgraph to the end node of $A_{i}$ and denote it by $A_{i}.level$. Now the upper bound of $g$'s average execution time $T_{avg}(g)$ can be defined as follows:

1. Construct simple parallel blocks $B_{i}$ such that if $A_{i} \in B_{i}$, then $A_{i}.level = i$, $i = 1, \ldots, M$ and $M$ is the maximum level of $g$.

2. Calculate $T_{oo}(B_{i})$ by using equations in section 4;

3. Calculate the upper bound:

$$TU_{oo}(g) = \sum_{i=1}^{M} T_{oo}(B_{i}).$$

Assertion 6. If $g$ is a nested transaction built by using serial and parallel block constructors, then

$$T_{avg}(g) \leq TU_{oo}(g) \leq T_{max}(g).$$

6 Results Analysis

Our first analysis is to find out how the results calculated from the formulas of the P-block evaluation in Section 4 compare with the results obtained from a simulation of the nondeterministic algorithm of Section 3.2.

Figure 2: Average execution time and simulation results

Figure 2 shows the comparison between calculated and simulated results. Here we set the number of flat transactions of the $P$-block $(m)$ to 40 and the number of database servers $(n)$ to 10. The number of shared data items $(l)$ changes from 1 to 40. The calculation result is shown in solid line and the simulation result is shown in line-points. It is seen that the two results are very close.

Figure 3: Average execution time (fixed l and n)

Our second analysis is to view how the changes of the number of flat transactions within a $P$-block will affect the execution time. Figure 3 shows the calculated
results when the number of shared data items (l) and the number of database servers (n) are fixed and the number of flat transactions (m) changes from 1 to 50. We have fixed the l and n in three situations. The solid line represents the expected execution time change when l = 5 and n = 5. The curve of errorbars represents the situation where l = 8 and n = 8. The curve of points represents the situation where l = 5 and n = 8.

Two observations can be drawn from Figure 3. Firstly, we can see the trend that the execution time increases when the number of flat transactions of the P.block increases. Also, the increment rate grows when m becomes larger. The second observation is that the l and n combinations also greatly affect the execution time.

![Figure 4: Average execution time (fixed m and n)](image)

The third analysis is to see how the changes of the number of shared data items affect the execution time. Figure 4 depicts the execution time changes when the number of flat transactions of the P.block (m) and the number of database servers (n) are fixed and the number of shared data items (l) changes from 1 to 40. The (m, n) combination has been fixed in three cases. The curve of errorbars shows the execution time where m = 20 and n = 5. The curve of solid line shows the situation where m = 20 and n = 8. The curve of points shows the situation where m = 10 and n = 8.

Figure 4 tells us that the execution time decreases sharply as the number of shared data items increases. But at a certain point, the execution time will no longer decrease much even the number of shared data items increases drastically. It also tells us that the (m, n) combinations greatly affect the execution time.

It is also very interesting to know how the transaction execution time changes along with the lower and upper bounds in general case. This is our fourth analysis. Here we use a complex nested transaction program, as shown in Figure 5, to show the comparison of the simulation results and the upper and lower bounds. The program has 54 flat transactions and 11 nodes (including start and finish nodes). For simplicity in simulation, the program does not include any S.block. Figure 6 shows the results. The upper bound is in solid line, the lower bound is in line-points, and the simulation results are represented as the curve of the errorbars. It is seen from this figure that the execution time and lower/upper bounds decrease drastically as the number of shared data items increases from 1 to 10. After that, the decrement of simulated time and lower/upper bounds is very slow. The simulated execution time keeps within the lower and upper boundaries as l changes. This is consistent with Assertions 5 and 6.

![Figure 5: Flowgraph of a nested transaction program](image)

![Figure 6: Simulation and lower and upper bounds](image)

References


An Algorithm for Estimating the Execution Time of Nested Transactions on A Locally Distributed Database System

Wanlei Zhou
School of Computing and Mathematics
Deakin University, Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract
This paper describes an algorithm for estimating the execution time of nested transactions running on a locally distributed database system. We first establish the model of nested transactions and the model of a locally distributed database system. Then we present a non-deterministic algorithm that can evaluate the execution time of a nested transaction using general routing strategies. The last part of the paper analyses the algorithm, presenting solutions to some special cases and the lower and upper bounds of executing time estimation.

Key Words: Nested transaction; Locally distributed database systems; Performance evaluation; Combinatorics.

Introduction
The demand for high transaction processing rate has motivated the development of multiprocessor or locally distributed database systems. A locally distributed database system has a tightly interconnection among its client-end system and the database servers, and therefore the communication delay is negligible compared to database transaction delays [5]. Some industrial and academic systems, such as the Tandem's Non-Stop system [2] and the Gamma database machine [1], have followed this architecture. Figure 1 depicts the architecture of a locally distributed database system. The client-end system and each database server are usually located on separate machines.

There have been some discussions on the execution time evaluation and routing strategies of transactions in a locally distributed database system. Examples are the minimum response time strategy proposed by Yu et al [5], the heuristic algorithm of Carey and Lu [1], and the dynamic routing strategies evaluated by Yu et al [6]. However, previous studies on the execution time estimation of nested transactions on locally distributed database systems have two limitations. Firstly they concentrate on response time estimation of some specific routing strategies, not general routing strategies. Secondly, they all use queueing theory to evaluate their models and therefore the result of the evaluation reflects the performance of a class of transactions and the system, not a specific transaction.

Our work is to study the execution time of nested transactions when

- general routing strategies, instead of specific routing strategies, are used,
- the user's viewpoint, instead of the system's viewpoint, is used in modeling the execution time of nested transactions, and
- routing of tasks to database servers (processors) as well as routing of tasks to shared data items are considered.

2 The Nested Transaction Model
2.1 Block Constructors
Definition 1. A flat transaction is a unit of database processing that cannot be divided and it can be allocated independently to a database server for processing. We use $A_{\text{subscript}}$ or $B_{\text{subscript}}$ to denote a flat transaction.

We use the following block constructors to build nested transactions.

- Serial block ($S\text{.block}$). A serial block is indicated by
BEGIN $A_1; A_2; \cdots; A_m$ END

where $A_i$'s are flat transactions and they are to be executed sequentially, i.e., $A_i$ must be executed before $A_{i+1}$ (i = 1, 2, \cdots, m-1). A serial block commits if all its flat transactions commit. We use $S(A_1, A_2, \cdots, A_m)$ to denote a serial block.

* **Parallel block (P.block)**. We define a parallel block as

\[
\text{COBEGIN } A_1; A_2; \cdots; A_m \text{ COEND}
\]

where $A_i$'s are flat transactions and they are to be executed concurrently. A parallel block commits if all its flat transactions commit. We use $P(A_1, A_2, \cdots, A_m)$ to denote a parallel block.

* **Selective block (L.block)**. We define a selective block as follows:

\[
\text{SEBEGIN } A_1; A_2; \cdots; A_m \text{ SEEND}
\]

where $A_i$'s are flat transactions and they are to be executed concurrently. The transaction identified by this selective block commits if one of its flat transaction commits. We use $L(A_1, A_2, \cdots, A_m)$ to denote a selective block.

Our model is concerned with nested transactions constructed from a set of flat transactions by the repeated application of the above block constructors.

**Definition 2.** A nested transaction is constructed through the application of the following abstract syntax:

\[
\text{NestedTrans ::= S\_block | P\_block} \\
\text{S\_block ::= BEGIN \{ NestedTrans \} END} \\
\text{P\_block ::= COBEGIN \{ NestedTrans \} COEND} \\
\text{L\_block ::= SEBEGIN \{ NestedTrans \} SEEND} \\
\text{Flat\_Trans ::= Flat transaction}
\]

An example of a nested transaction constructed by the above blocks is listed below:

BEGIN $A_1$; 
COBEGIN 
BEGIN 
  $A_2$; COBEGIN $A_{21}; A_{22}$ COEND 
END 
$A_3$; BEGIN 
$A_4$; COBEGIN $A_{41}; A_{42}; A_{43}; A_{44}$ COEND 
END 
SEBEGIN $A_{51}; A_{52}; A_{53}$ SEEND 
COEND 
$A_6$ 
END

\[ (2.1) \]

If a block $X$ is inside another block $Y$, we call $X$ a member of $Y$. In the listing (2.1), flat transactions $A_{21}$ and $A_{22}$ are direct members of a parallel block $X$: $P(A_{21}, A_{22})$. The flat transaction $A_2$ and the parallel block $X$ are again the direct members of the serial block $Y$: $S(A_2, P(A_{21}, A_{22}))$. $A_{21}$ and $A_{22}$ are called the non-direct members of $Y$ if we want to distinguish them from direct members.

If all direct members of a (serial, parallel, or selective) block are flat transactions, we call this block a simple block. So, $X$ is a simple parallel block whereas $Y$ is not a simple block. Simple blocks are easier to evaluate.

### 2.2 Flowgraphs

Equivalently we can associate each nested transaction with a flowgraph. The nodes of the graph represent the starting and synchronisation points of flat transactions and block constructors, and the arcs of the graph represent flat transactions. Two special nodes, denoted by $s$ and $f$, indicate the starting point and finish point of the nested transaction, respectively. The flowgraphs of the block constructors are indicated by Figure 2, where Figure 2(a), (b) and (c) represent the flowgraphs of the serial block, the parallel block and the selective block, respectively.

We use two adjacent nodes to represent situations where two transaction blocks share a common start or finish node. For example, Figure 3(a) depicts the following nested transaction:

\[ P(A_1, \cdots, A_i, L(B_1, \cdots, B_j), A_{i+1}, \cdots, A_m) \]  \[ (2.2) \]
By repeated application of the nested transaction block constructors, one can associate with each nested transaction an equivalent program flowgraph. Figure (b) depicts the flowgraph of listing (2.1).

Assertion 1. The flowgraph of a nested transaction is acyclic.

Proof: If there is a cycle, there must be a block which is a member of itself. This is impossible because each time one applies a block constructor to one or several blocks, those blocks become the direct member of higher level block. That is, a block cannot have itself as one of its members.

The Evaluation Model

1. The Locally Distributed Database System

The locally distributed database system studied in this paper has a similar architecture as the system described in [6]. The system has \( n \) database servers and one front-end system that is connected via a high-speed interconnection network. Database servers are denoted by \( S_i \) and the databases managed by these database servers are denoted by \( DB_i \), where \( i = 1, 2, \ldots, n \), respectively. All database requests to \( DB_i \) are assumed to be handled by the database server \( S_i \).

Transactions submitted by users enter the system through the front-end system where a routing strategy is employed to route the incoming transactions to database servers and to shared data items. At the assigned database server, a transaction is served if the shared data item it requires is not in use. Otherwise, the transaction is queued up in a waiting queue until the shared data item is free.

A routing strategy can be represented as a function \( A \rightarrow S \) where \( A \) is the set of flat transactions of a nested transaction and \( S \) is the set of database servers: \( A = \{ A_1, A_2, \ldots, A_m \} \) and \( S = \{ S_1, S_2, \ldots, S_n \} \). Further, when a flat transaction \( A_i \) is routed to a database server \( S_i \), we use another function \( v : A \rightarrow D \) to represent the mapping of \( A_i \)'s to particular shared data items. Here \( D \) is the set of shared data items in database server \( S_i \): \( D = \{ D_1, D_2, \ldots, D_m \} \).

Let us model the required execution time of a flat transaction by a function \( t : A \rightarrow \text{real numbers} \). This required execution time is the execution time where the at transaction does not wait for any shared data item and therefore is executed immediately after it is exclusively routed to a database server. Given a flat transaction \( A_i \in A \), the front-end system tells us that \( A_i \) is going to be executed on the database server \( u(A_i) \), that \( A_i \) is going to use a shared data item \( v(A_i) \), and that the execution of \( A_i \) will require time \( t(A_i) \).

If \( k \) flat transactions \( A_i, i = 1, \ldots, k \) are routed to different database server at the same time and they all use different shared data items, then these flat transactions can be executed on the database server simultaneously. Because they share the same database server, their execution time will be greater than the required execution time \( t(A_i) \). It is expected that real execution time will increase if \( k \) increases [4]. We use a function \( f(k) \) to represent this slow-down factor and denote the execution time of each \( A_i \) by \( t(A_i) \times f(k) \). It is reasonable to assume that \( f(k) = 1 \) when \( k = 1 \) and \( 1 \leq f(k) \leq k \) when \( k > 1 \).

3.2 The Nondeterministic Algorithm

We use a nondeterministic algorithm to evaluate the execution time of a nested transaction. The algorithm simulates the execution of the nested transaction and collects the execution time along the way of the execution. The non-determinism is reflected by the routing strategy adopted by the algorithm during the execution: different routing strategies may result different execution times for the same nested transaction. The algorithm uses a "marking scheme" on the associated flowgraph of the nested transaction. Each arc (flat transaction) can be marked as one of the following (where \( S_i \) represents the \( i \)th database server and the \( D_j \) represents the \( j \)th shared data item of database server \( S_i \)):

<table>
<thead>
<tr>
<th>Marked State</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>unmarked</td>
<td>waiting to be routed by the front-end system</td>
</tr>
<tr>
<td>allocated</td>
<td>allocated to ( S_i ) and to use ( D_j )</td>
</tr>
<tr>
<td>executing</td>
<td>( S_i ) is executing the flat transaction and it uses ( D_j )</td>
</tr>
<tr>
<td>completed</td>
<td>the execution of the flat transaction is completed</td>
</tr>
</tbody>
</table>

All arcs progress through the marks in this order. The marking scheme has the following invariant:

All arcs marked "executing" are associated with a different database server, or are associated with a different shared data item within the same database server.

Further, we say that an arc marked "allocated" can execute if no arc marked "executing" is associated with the same database server, or all arcs marked "executing" and associated with the same database server are not associated with the same shared data item.

To evaluate the execution time, we associate with each node and each arc of the flowgraph an attribute called CompletionTime. We also use a single global variable denoted CurrentTime (with initial value 0) to record the current time.

The algorithm is listed below:

1. Initialisation:
2. Assign CompletionTime and CurrentTime to 0 at start node;
3. Mark all arcs $A_i$ leaving start node "allocated $(u(A_i), v(A_i))$";
4. while there exist allocated arcs which can execute {
5. select an allocated arc $A_i$ which can execute;
6. mark it "executing";
7. $A_i$.CompletionTime = CurrentTime+$t(A_i) \times f(k)$;
8. }
9. while there exist arcs marked "executing" {
10. select an "executing" arc $A_i$ with minimum $A_i$.CompletionTime;
11. mark it "completed";
12. CurrentTime = $A_i$.CompletionTime;
13. denote target node of $A_i$ by $y$;
14. if the arc just marked belongs to a selective block {
15. mark all arcs of that selective block "completed";
16. set all CompleteTime of these arcs to CurrentTime;
17. }
18. If all arcs $A_i$ entering $y$ are marked "complete" {
19. $y$.CompletionTime = max$(A_i$.CompletionTime$)$;
20. mark "allocated $(u(A_i), v(A_i))$" to all arcs leaving $y$;
21. }
22. while there exist allocated arcs which can execute {
23. select an allocated arc $A_i$ which can execute;
24. mark it "executing";
25. $A_i$.CompletionTime = CurrentTime+$t(A_i) \times f(k)$;
26. }
27. }
28. The nested transaction execution time is the CompletionTime of the finish node;

$$T_{max}(g) = \max_{u \in U, v \in V}(T_{u,v}(g)),$$

$$T_{avg}(g) = \frac{\sum_{u \in U, v \in V} \text{pr}(u,v) \times T_{u,v}(g)}{\sum_{u \in U, v \in V} \text{pr}(u,v)},$$

where $\text{pr}$ is a probability function defined over the sets of maps $U$ and $V$. We are only interested in the equal-probable case, that is, all routing strategies are treated equally. So we have,

$$\text{pr}(u,v) = \begin{cases} \frac{1}{|E|} & \text{for } U, \text{ where } E = \text{card}(E) \\ \frac{1}{|F|} & \text{for } V, \text{ where } F = \text{card}(F) \end{cases}$$

4.2 Serial Blocks

It is easy to estimate the execution time of a serial block by simply adding the execution times of all its direct members. That is, let $S$ be a serial block defined as $S(A_1, A_2, \cdots, A_m)$, where $A_i$'s ($i = 1, 2, \cdots, m$) are $S$'s direct member blocks. Let $T(A_i)$ ($i = 1, 2, \cdots, m$) be the estimated execution times of those members, then the execution time of $S$ is

$$T_S = \sum_{i=1}^{m} T(A_i).$$

4.3 Analysis of Simple P Blocks

The nested transaction of a simple parallel block has the following structure:

$$P(S_1, A_1, A_2, \cdots, A_m)$$

where $A_i$'s are flat transactions. We assume that $f(k) = k, k \geq 1$. We also assume that all flat transactions have the same required execution time:

$$t(A_i) = 1, \forall A_i \in C.$$
while the time needed to execute all queues on \( S_i \) is the
of the largest queue in \( S_i \):

\[
T_{u,v}(C_{S_i}) = \max_{j \in C_{S_i}}(T_{u,v}(C_{S_i}, D_{ij})).
\]  

(8)

then we can express the time needed to execute all flat
actions of a parallel block as follows:

\[
T_u,v(g) = \max_{j \in C_{S_i}}T_u,(C_{S_i}, D_{ij}).
\]  

(9)

cause of assumption of (6), equation (7) can be writ-

\[
T_u,v(C_{S_i}, D_{ij}) = \text{card}(C_{S_i}, D_{ij}).
\]  

(10)

Solution for Simple P. Blocks

We characterise the sets \( U \) and \( V \) of routing strategy
ps as follows. Each \( u \in U \) is a map from the finite
\( C \) to the finite set \( S \) and each \( v \in V \) is a map from
finite set \( C \) to the finite set \( D \) where we denote

\[
i = \text{card}(S), m = \text{card}(C), \text{ and } l = \text{card}(D).
\]  

(11)

When maps are being set up in the front-end system.
flat transactions themselves are not distinguished.
their routing to a database server that is of interest.
we identify a distinct routing map \( u \) with the alloca-
\( \text{tion of } m \text{ identical "balls" to } n \text{ different \"boxes"}, \text{ and \"
tinct routing map } v \text{ with the allocation of } C, \text{ identi-
\( \text{l "balls" to } l \text{ different \"boxes"}. \text{ This is a standard
blem in combinatorics. The maximum and minimum
asures are easy. We have:}

Assertion 5. If \( g \) is a simple parallel block, then

\[
T_{\text{max}}(g) = m
\]  

(12)

\[
T_{\text{min}}(g) = \left\lceil \frac{m}{l} \right\rceil
\]  

(13)

if: Equation (12) corresponds to the case where all
transactions are mapped to a single database server
then mapped to a single shared data item. This
idently the worst case. Equation (13) corresponds
he case where the flat transactions are distributed
enly as possible to all the database servers and the
data items. This is evidently the best case. Here
denotes the integer least upper bound.
et assume that \( x \) flat transactions of \( C \) are routed
database server, i.e., \( x = \text{card}(C_{S_i}) \). Because
action time of this situation has nothing to do
the database server \( S_i \) (we assume that all
base servers are identical) but it is related to the
of \( x \), we can then denote \( T_{\text{avg}}(C_{S_i}) = T_{\text{avg}}(x) \).
ne now compute \( E \), the total number of maps which \( x \)
(identical) flat transactions into \( l \) (identical)
data items. From combinatorics results we know
is a type-2 distribution problem and that

\[
E = d^2_x(l, [0, \infty)) = \left( \begin{array}{c} l + x - 1 \\ l - 1 \end{array} \right)
\]  

(14)

where \( [0, \infty) \) is the restrict condition of flat transactions
in a single shared data item. But from the problem we
we know the maximum number of flat transactions within
a single database server can not be more than \( x \). So
here the restrict condition can be changed to \([0, x]\).

If we denote by \( Q^x_{S_i} \) the number of maps with
\( \max_{j \in C_{S_i}} \text{card}(C_{S_i}) = y \), then

\[
T_{\text{avg}}(x) = \frac{1}{E} \sum_{y=1}^{\infty} y \times Q^x_{S_i}
\]  

(15)

The combinatorial meaning of \( Q^x_{S_i} \) is: \( x \) identical flat
actions are going to use \( l \) different shared data items and the maximum number of flat transactions
using a single shared data item is exactly equal to \( y \)
and at least one shared data item has this number of
flat transactions using it. If the maximum number of
flat transactions using a single shared data item is less
than or equal to \( y \), then the number of maps under this
restrict condition is

\[
d^2_x(l, [0, y]) = \sum_{0 \leq y \leq l} (-1)^y \left( \begin{array}{c} l \\ y \end{array} \right) \left( \begin{array}{c} l + y - (y - 1)z - 1 \\ l - 1 \end{array} \right)
\]  

So,

\[
Q^x_{S_i} = d^2_x(l, [0, y]) - d^2_x(l, [0, y - 1])
\]  

(16)

\( y = 1, 2, \ldots, x \) and \( d^2_x(l, [0, 0]) = 0 \).

It is evident that

\[
E = \sum_{y=1}^{\infty} Q^x_{S_i}
\]  

(17)

So we have the following:

Assertion 6. If \( x \) flat transactions are routed to a
data base server \( S_i \), then the average execution time of
all these \( x \) flat transactions is:

\[
T_{\text{avg}}(x) = \frac{1}{E} \sum_{y=1}^{\infty} y \times Q^x_{S_i}
\]  

(17)

where \( E \) and \( Q^x_{S_i} \) are defined in equations (14) and 16,
respectively.

Now let us compute \( T_{\text{avg}}(g) \). Let \( F \) be the number
of maps which map \( m \) (identical) flat transactions into
\( n \) (identical) database servers. This is again a a type-2
distribution problem and

\[
F = d^2_m(n, [0, \infty)) = \left( \begin{array}{c} n + m - 1 \\ n - 1 \end{array} \right)
\]  

(18)

By a similar analysis as we have done for obtaining
\( T_{\text{avg}}(x) \), we have
Assertion 7. If \( g \) is a simple parallel block, then the average execution time of \( g \) is

\[
T_{\text{avg}}(g) = \frac{1}{F} \sum_{x=1}^{x=n} z \times Q_{2}^{m/n} \times T_{\text{avg}}(x). \tag{19}
\]

5 Lower and Upper Bounds Estimation

In this section we try to estimate the lower and upper bounds of execution time of a general nested transaction. Here the lower bound does not mean that any execution time of a nested transaction must be greater than it. It is obvious that some of the execution time evaluated from some special scheduling strategies may be less than the lower bound. But we say that the average estimated execution time of a nested transaction will certainly be greater than the lower bound. Similar claim holds for the upper bound. We omit the proofs of the two assertions on the correctness of lower and upper bounds estimation.

5.1 Lower Bound

We define an extended parallel block as

\[
\text{COBEGIN } E_{1}; E_{2}; \ldots; E_{k} \text{ COEND}
\]

where \( E_{i} \) is a simple parallel block, or a simple selective block, or an extended parallel block. This is a recursive definition.

We now simplify the execution time estimation of an extended parallel block to a feasible form. If there are \( n \) database servers and \( l \) shared data items, we define the lower bound of the average execution time of an extended parallel block \( A \) as in

\[
T_{\text{avg}}(A) = \max_{i=1}^{l}(T_{\text{avg}}(E_{i})). \tag{20}
\]

If \( E_{i} \) is a simple parallel block, then \( T_{\text{avg}}(E_{i}) \) is calculated by using the equations in Section 4.4. That is, in this case we have \( T_{\text{avg}}(E_{i}) = T_{\text{avg}}(E_{i}) \). If \( E_{i} \) is a simple selective block, then \( T_{\text{avg}}(E_{i}) \) is the smallest required execution time of its member flat transactions. Otherwise, \( T_{\text{avg}}(E_{i}) \) is calculated by using equation (20) again. So the calculation is also recursive. That also explains the feasibility of calculating equation (20).

By extending the definition of equation (20) to a nested transaction \( g \), we can have the lower bound execution time of \( g \). The calculation can be carried out from the inner simple blocks of \( g \) to the outer blocks. For a parallel block, equation (20) is applied, for a selective block, the minimum execution time of its members is selected, while for a sequential block, equation (4) of Section 4.2 is applied. The claim that \( T_{\text{avg}}(A) \) is a lower bound is justified by the following result.

**Assertion 8.** If \( g \) is a nested transaction, then

\[
T_{\text{min}}(g) \leq T_{\text{avg}}(g) \leq T_{\text{avg}}(g). \tag{21}
\]

5.2 Upper Bound

If \( g \) is a nested transaction, we define the level of a flat transaction \( A_{i} \in g \) as the length of the longest path from the start node \( s \) of the nested transaction flowgraph to the end node of \( A_{i} \) and denote it by \( A_{i, \text{level}} \). Now the upper bound of \( g \)'s average execution time \( TU_{\text{avg}}(g) \) can be defined as follows:

1. Construct simple parallel blocks \( B_{i} \) such that if \( A_{i} \in B_{i} \), then \( A_{i, \text{level}} = i \), \( i = 1, \ldots, M \) and \( M \) is the maximum level of \( g \).
2. If \( B_{i} \) includes any \( A_{j} \) from a selective block, then set \( T_{\text{avg}}(B_{i}) \) to the maximum of all \( T_{\text{avg}}(A_{j}) \). Otherwise, calculate \( T_{\text{avg}}(B_{i}) \) by using equations in Section 4.4.
3. Calculate the upper bound:

\[
TU_{\text{avg}}(g) = \sum_{i=1}^{M} T_{\text{avg}}(B_{i}).
\]

It is easy to see that the calculation of the upper bound is feasible. The claim that \( TU_{\text{avg}}(g) \) is indeed an upper bound is justified by the following result.

**Assertion 9.** If \( g \) is a nested transaction, then

\[
T_{\text{avg}}(g) \leq TU_{\text{avg}}(g) \leq T_{\text{max}}(g). \tag{22}
\]

6 Results Analysis

Our first analysis is to find out how the results calculated from the formulas of the P-block evaluation in Section 4.4 compare with the results obtained from a simulation of the nondeterministic algorithm of Section 3.2.

Figure 4 shows the comparison between calculated and simulated results. Here we set the number of flat transactions of the P-block (m) to 40 and the number of database servers (n) to 10. The number of shared data items (l) changes from 1 to 40. The calculation result is shown in solid line and the simulation result is shown in line-points. It is seen that the two results are very close.

During the simulation, our simulation program runs 1000 times for each \( (l, m, n) \) combination. Each combination represents a scheduler mapping which maps \( m \) flat transactions to \( n \) database servers and then maps those flat transactions that have been mapped to a database server to \( l \) shared data items. The average values of these 1000 simulation runs is the execution time of this particular combination.

Our second analysis is to view how the changes of the number of flat transactions within a P-block will affect the execution time. Figure 5 shows the calculated results when the number of shared data items (l) and
Figure 4: Average execution time and simulation results

Figure 5: Average execution time when l and n are fixed

Figure 6: Average execution time when m and n are fixed

The third analysis is to see how the changes of the number of shared data items affect the execution time.

Figure 6 depicts the execution time changes when the number of flat transactions of the F.block (m) and the number of database servers (n) are fixed and the number of shared data items (l) changes from 1 to 40. The (m, n) combination has been fixed in three cases. The curve of errorbars shows the execution time where m = 20 and n = 5. The curve of solid line shows the situation where m = 20 and n = 8. The curve of points shows the situation where m = 10 and n = 8.

Figure 6 tells us that the execution time decreases sharply as the number of shared data items increases. But at a certain point, the execution time will no longer decrease much even the number of shared data items increases drastically. It also tells us that the (m, n) combinations greatly affect the execution time.

It is also very interesting to know how the transaction execution time changes along with the lower and upper bounds in general cases. This is our fourth analysis. Here we use a complex nested transaction program, as shown in Figure 7, to show the comparison of the simulation results and the upper and lower bounds. The program has 54 flat transactions and 11 nodes (including start and finish nodes). For simplicity in simulation, the program does not include any S.block.

Figure 7: Flowgraph of a complex nested transaction program

Our purpose here is to show the variation of the program's execution as well as lower and upper bounds when the number of shared data items changes. It is not feasible to obtain the explicit solution for the program because of its complex structure. So we use the algorithm of Section 3.2 to simulate the execution time of the program. We also use 1000 runs for each situation, as we have done before, and then we average the results of these 1000 runs to get a simulation result. Figure 8 shows the results. The upper bound is in solid line, the lower bound is in line-points, and the simulation results are represented as the curve of the errorbars. It is seen from this figure that the execution time and lower/upper bounds decrease drastically as the number of shared data items increases from 1 to 10. After that, the decrement of simulated time and
lower/upper bounds is very slow. The simulated execution time keeps within the lower and upper boundaries as \( t \) changes. This is consistent with Assertions 8 and 9.

![Graph showing execution time bounds](image)

Figure 8: Simulated execution time and lower and upper bounds when \( m \) and \( n \) are fixed

7 Remarks

An algorithm for the evaluation of the execution time of a nested transaction on a locally distributed database system is developed in this paper. The algorithm considers all the factors such as the generality of routing strategies, the sharing of database servers, and the sharing of shared data items. We evaluate this execution time from the user's viewpoint instead of from the system's viewpoint, because a programmer is usually interested in his/her own program instead of the whole system.

References


Supporting fault-tolerant and open distributed processing using RPC

Wanlei Zhou*

School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia

Received 28 October 1994; revised 2 April 1995

Abstract

This paper is concerned mainly with the software aspects of achieving reliable operations in an open distributed processing environment. A system for supporting fault-tolerant and cross-transport protocol distributed software development is described. The fault-tolerant technique used is a variation of the recovery blocks and the distributed computing model used is the remote procedure call (RPC) model. The system incorporates fault tolerance features and cross-transport protocol communication features into the RPC system and makes them transparent to users. A buddy is set up for a fault-tolerant server to be its alternative. When an RPC to a server fails, the system will automatically switch to the buddy to seek for an alternate service. The client, the fault-tolerant server and the buddy of the server can all use a different transport protocol. To obtain this fault tolerance and cross-protocol service, users only need to specify their requirements in a descriptive interface definition language. All the maintenance of fault tolerance and the cross-protocol communication is managed by the system in a user transparent manner. By using our system, users will have confidence in their distributed programs without bothering the fault tolerance and cross-protocol communication details. Our system is small, simple, easy to use and also has the advantage of producing server and client driver programs, and finally, executable programs directly from the server definition files.

Keywords: Open distributed processing; Fault-tolerant computing; Distributed systems; Remote procedure calls; Client/server model

1. Introduction

The advances in computer technology have made it cost-effective to build distributed systems in various applications. Many experts agree that the future of distributed computing, especially the future of open distributed processing, is the future of computing. The network is the computer has become a popular phrase [1].

Remote Procedure Call (RPC) is perhaps the most popular model used in today's distributed software development, and has become a de facto standard for distributed computing. To use it in an open distributed environment effectively, however, one has to consider cross-protocol communications, because user programs built on top of different RPC systems cannot be interconnected directly. Typical solutions to this problem are:

1. Black protocol boxes: protocols used by RPC programs are left as black boxes in compiling time, and are dynamically determined in binding time [2].

However, one issue is still outstanding in building RPC systems for open distributed systems; fault-tolerance features.

An open distributed system consists of many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever increasing dependency being placed on open distributed systems, the number of users requiring fault tolerance is likely to increase.

The design and understanding of fault-tolerant open distributed systems is a very difficult task. We have to deal with not only all the complex problems of open distributed systems when all the components are well, but also the more complex problems when some of the components fail.
This paper is concerned mainly with the software aspects of achieving reliable operations in an open distributed processing environment. A system for supporting fault-tolerant and cross-transport protocol distributed software development is described. The system design is aimed towards application areas that may involve a heterogeneous environment, and in which requirements for fault-tolerance are less severe than in, for example, the aerospace field, but in which continuous availability is required in the case of some components failures [5]. The application areas could be, for example, kernel/service pool-based distributed operating systems, supervisory and telecontrol systems, switching systems, process control and data processing. Such systems usually have redundant hardware resources, and one of the main purposes of our system is to manage the software redundant resources in order to exploit the hardware redundancy. The fault-tolerant technique used is a variation of the recovery blocks technique, and the distributed computing model used is the RPC model.

Software fault tolerance refers to the set of techniques for continuing service despite the presence, and even the manifestation, of faults in a program [6]. There are many techniques available for software fault-tolerance, such as N-version programming [7] and recovery blocks [8]. In N-version programming, \( N(N \geq 2) \) independent and functionally equivalent versions of a program are used to process a critical computation. The results of these independent versions are compared (usually with a majority voting if \( N \) is odd) at the end of each computation, and a decision will be made accordingly.

Recovery blocks employ temporal redundancy and software standby sparing [9]. Software is partitioned into several self-contained modules called recovery blocks. Each recovery block consists of a primary routine, which executes critical software function; one or more alternate routines, which performs the same function as the primary routine, and is invoked upon a failure is detected; and an acceptance test, which tests the output of the primary (and alternate, if the primary fails) routine after each execution. A variation of this model is used in this paper.

The remote procedure call is a powerful and widely known primitive for distributed programming [10]. The RPC based model allows a programmer to call a procedure located at a remote computer in the same manner in which a local procedure is called. This model has a lot of advantages. The procedure call is a widely accepted, used and understood abstraction. This abstraction is the sole mechanism for accessing remote services in this model. So the interface of a remote service is easily understood by any programmer with a good knowledge of ordinary programming languages.

The remainder of this paper is organised as follows. In Section 2, we summarize some notable related work and provide the rationale of our work. In Section 3, we describe the architecture of the SRPC system. Then Section 4 describes the syntax and semantics of the server definition files and the stub and driver generator. In Section 5 we present an example to show how this system can be used in supporting fault-tolerant, open distributed software development. Section 6 contains remarks.

2. Related work and the rationale

There have been many successful RPC systems since Nelson's work [11], such as Cedar RPC [12], NCA/RPC [13], Sun/RPC [14], HRPC [2], and so on. But few of them consider fault tolerance an cross-protocol communication in their design, or they rely on users to build in these features.

Notable work on incorporating fault tolerance features into RPC systems is Argus [15], ISIS [16,17] and an atomic RPC system on ZMOB [18]. The Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. Atomic transactions are serializable and indivisible. A user can also define some atomic objects, such as atomic arrays and an atomic record, to provide the additional support needed for atomicity. All the user fault tolerance requirements must be specified in the Argus language.

The ISIS toolkit is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. A special process group called a fault tolerant group, is established when a group of processes (servers and clients) are cooperating to perform a distributed computation. Processes in this group can monitor one another, and can then take actions based on failures, recoveries or changes in the status of group members. A collection of reliable multicast protocols is used in ISIS to provide failure atomicity and message ordering.

The atomic RPC system implemented on ZMOB uses sequence numbers and calling paths to control the concurrency and atomicity, and used checkpointing to maintain the ability of recovering from failures. Users do not have to provide synchronization and recovery themselves; they only need to specify if atomicity is desired. This frees them from managing much complexity.

But when a server (or a guardian in the Argus) fails to function well, an atomic transaction or an atomic RPC has to be aborted in these systems. This is a violation of our continuous computation requirement. The fault-tolerant process groups of the ISIS can cope with process failures and can maintain continuous computation, but the ISIS toolkit is big and relatively complex to use.
Typical solutions to the cross-protocol communication in RPC systems are the black protocol boxes of the HRPC [2], the special protocol conversion interface 3 and the RPC agent synthesis system [4] for cross-RPC communications.

The HRPC system defines five RPC components: the stub, the binding protocol, the data representation, the transport protocol and the control protocol. An HRPC client or server and its associated stub can view each of the remaining components as a black box. These black boxes can be 'mixed and matched'. The set of protocols to be used is determined at bind time - long after the client and server has been written, the stub has been generated, and the two have been linked.

The special protocol conversion interface that we proposed earlier [3] uses an 'interface server' to receive a call from the source RPC component (client or server) and to convert it into the call format understood by the destination RPC component (server or client).

The cross-RPC communication agent synthesis system [1] associates a 'client agent' with the client program and a 'server agent' with the server program. A link protocol is then defined between the two agents and allows them to communicate. The server and the client programs can use different RPC protocols, and the associated agents will be responsible for converting these dialect protocols into the link protocol.

But none of the above cross-protocol RPC systems consider fault-tolerance issues. If the server fails, the client simply fails as well.

Incorporating both fault tolerance and cross-protocol communication into RPC systems is clearly an important issue to use RPCs efficiently and reliably in open distributed environments. In this paper we describe a system, called the SRPC (Simple RPC) system, for supporting the development of fault-tolerant, open distributed software. The SRPC incorporates fault tolerance features and protocol converters into the system and makes them transparent to users.

'buddy' is set up for a fault-tolerant server to be its alternate. When an RPC to a server fails, the system automatically switches to the buddy to seek for an alternate service. The RPC aborts only when both the server and its buddy fail. The clients and servers can use different communication protocols. To obtain these fault tolerance and automatic protocol conversion, etrs only need to specify their requirements in a descriptive interface definition language. All the management of fault tolerance and protocol conversion are managed by the system in a user transparent manner. By using our system, users will have confidence on their open distributed computing without bothering the fault tolerance tasks and protocol conversion. Our system is small, simple, easy to use and also has the advantage of producing server and client programs and finally executable programs directly from the server definition files.

3. System architecture

The SRPC is a simple, fault-tolerant and cross-protocol remote procedure call system [19]. The system is small, simple, expandable, and it has facilities supporting fault-tolerant computing and cross-protocol communication. It is easy to understand and easy to use. The SRPC only contains the essential features of an RPC system, such as a location server and a stub generator, among other things. The SRPC system has been used as a distributed programming tool in both teaching and research projects for three years.

The SRPC system has another interesting feature. That is, the stub compiler (we call it the stub and driver generator, or SDG in short) not only produces the server and client stubs, but also creates remote procedures' framework, makesfile, and driver programs for both server and client. After using the make utility, a user can test the program's executability by simply executing the two driver programs. This feature will be more attractive when a programmer is doing prototyping.

3.1. Server types

The client/server model [20] is used in the SRPC system. An SRPC program has two parts: a server part and a client part. Usually the server provides a special service or manages an object. The client requests the service or accesses the object by using the remote procedures exported by the server.

There are three types of servers in the SRPC system: simply servers, service providing servers and object managing servers. Fig.1 depicts these three types of servers.

A simple server (Fig.1(a)) is an ordinary server possessing no fault-tolerant features. When a simple server fails, all RPCs to it have to be aborted.

A service providing server (Fig.1(b)) has a buddy server running somewhere in the network (usually on a host different from the server's), but no communication between the server and its buddy. When a service providing server fails, an RPC to this server will be automatically redirected to its buddy server by the system. As object changes in the server will not be available in its buddy, a service providing server usually is used in applications such as pure computation, information retrieval (no update), motor-driven (no action memory), and so on. It is not suitable to be used to manage any critical object that might be updated and then shared by clients.

An object managing server (Fig.1(c)) also has a buddy running in the network. It manages a critical object that might be updated and shared among clients. An RPC to such a server, if it will change the object state, is actually a nested RPC. That is, when the server receives such a call from a client, it first checks to see whether the call can be
executed successfully (e.g. if the necessary write-locks have been obtained or not). If the answer is no, the call is aborted. If the answer is yes, then the server will call its buddy server to perform the operation as well. When the buddy returns successfully, the call commits (the server and its buddy actually perform the call) and the result returns to the client. To ensure the consistency of the objects managed by the server and its buddy, a two-phase commit protocol [21] is used when executing the nested RPC.

Like a service providing server, when an object managing server fails, an RPC to this server will be automatically redirected to its buddy server by the system.

All buddy servers are simple servers. That means, when a server (service providing or object managing) fails, its buddy server provides alternative service in a simple server manner. Also, when a buddy server fails, a service providing server or an object managing server will be reduced into a simple server.

3.2. Architecture

The SRPC has the following three components: a Location Server (LS) and its buddy (LS buddy), a system library, and a Stub and Driver Generator (SDG). This section describes the system architecture from a user's point of view. As server buddies are generally transparent to users, we will omit their descriptions here.

From a programmer's viewpoint, after the SDG compilation (see Section 5), the server part of an SRPC program is consisted of a server driver, a server stub and a file which implements all the remote procedures (called a procedure file). The server buddies are transparent to users. The server part (or a server program as it is sometimes called) is a 'forever' running program which resides on a host and awaits calls from client. The client part (or a client program) consists of a client driver and a client stub after the SDG compilation. It runs on a host (usually a different host from the server's host), and makes calls to the server by using the remote procedure exported by the server.

When the client driver makes a call, it goes to the client stub. The client stub then, through the system library, makes use of the client protocol for sending the call to the server host. Because the client and server may use different communication protocols, client-server protocol converter is used to convert the client's protocol into server's protocol. The call message is then sent to the server. At the server's host, the server's protocol entity will pass the call message to the server stub through the system library. The server stub then reports the call to the server driver and an appropriate procedure defined in the procedure file is executed. The result of the call follows the call route reversely, through the server stub, the server protocol, the system library of the server host, the client server protocol converter, the system library of the client host, the client stub, back to the client driver. This is called a direct call, as the pre-condition of such a call is that the client knows the address of the server before the call.

With the help of the Location Server, the run-time address of a server can be easily accessed. One typical scenario of SRPC programs using LS can be describe below:

1. Registering: when the server is started, it first registers its location to the LS. The route is: server driver → server stub → server protocol, server-L protocol converter and the system library of the server host → LS protocol and system library of the L host → LS stub → LS driver.
2. Waiting: the server waits for client calls.
3. Locating: when a client is invoked, it consults the LS for the server's location. The route is: client driver → client stub → client protocol, client-L protocol converter and system library of the client host → LS protocol and system library of the L host → LS stub → LS driver.
RPC: after the location is found, the client then can make any number of RPCs to that server by using the obtained location (as in a direct call). We name this a typical calling, since most of the time a client does not know server addresses.

Shutdowm: if a 'shutdown' call is issued by a client program, it causes the server to un-register itself from the LS and exits from the system.

Fig. 2 depicts the system architecture using a typical PC. The dashed line represents the RPCs from the server's viewpoint.

3. Location server

One way of hiding out the implementation details is the use of the Location Server (LS). The LS is used to hide the server locations from users. It maintains a database of server locations and is executed before any other RPC program is started. After that, it resides on the host and awaits calls from servers and clients.

The Location Server is an object managing server and as a buddy of its own. It has a well-known location, and its location can be easily changed when necessary. The S itself is implemented by the SRPC system, using the rect calling method.

Usually there should be one LS (called local LS) running on each host for managing locations of that host, and these local LSs report to the 'global LS' (like the NCA/RPC's local and global location brokers) [13,22]. In that case, the locations of all LSs can also be hidden from users. We have planned to implement this facility.

The following call is used by a server to register itself to the LS:

```c
int registerServer(sn, buddy, imp)
char *sn;    /* server name */
char *buddy; /* buddy's name */
struct iinfo /* implementation */
*imp;       info,*
```

where imp is a type struct iinfo structure and contains many implementation details, such as the server's host name, protocol, and so on. Because the call updates the LS database, it is also directed to the LS buddy. If the call returns OK, the location has been registered and a client can use the following call to find out the location of a server from the LS:

```c
int locateServer(sn, buddy, imp)
char *sn;    /* server name */
char *buddy; /* server's buddy name */
struct iinfo /* implementation */
*imp;       info,*
```
If the call returns OK, the location of the server sn is stored in imp and the name of the server's buddy is stored in buddy for later use. This call does not affect the LS database state, so there is no hidden LS server and LS buddy communication here. Before a server is shut down, the following call must be used to un-register the server from the LS:

```c
int unregisterServer(sn)
char *sn;  /* server name */
```

If the call returns OK, the server and its buddy (if any) are deleted from the LS database. The system also provides other LS calls for maintaining the LS database.

All the usages of these functions in a server or a client program are automatically generated by the stub and server generator. A user does not need to look into the details of these calls if he or she is satisfied with the generated program sections.

### 3.4. System library

The system library is another way of achieving transparency. The library contains all the low-level and system- and protocol-oriented calls. Its main function is to make the low-level facilities transparent to the upper-level programs. So the stub and driver programs of both server and client will not deal with their communication entities directly.

The server and client programs must be linked with the system library separately. Ref. [19] contains detailed descriptions of the library calls. All the library calls can be divided into the following call levels, and Fig. 3 depicts their relationships:

1. SRPC Level: this is the highest level. It contains calls that deal with RPC-related operations.
2. Remote Operation Level: contains calls that deal with remote operations. These remote operations follow the definitions of the OSI Application level primitives [23].
4. Utility calls: contains all the utility calls used in different levels.

![Fig. 3 Relationships of system library levels.](image)

The innermost level is the protocol-specific level. It interfaces with the specific protocol entity and the underlying operating system kernel. It is also responsible for providing protocol converting procedures. The remote operation level provides a uniform interface (similar to the OSI Application level primitives) to the upper RPC system. The uniform interface provides two obvious advantages:

- It provides a clear interface for different communication protocols.
- It makes the SRPC system as portable as possible: only the lower levels need to be re-programmed when porting the SRPC system to other platforms.

The SRPC level implements all the RPC related calls and provides a user-friendly remote procedure calling interface to application programs. The utility calls provide service calls for different levels.

### 3.5. Performance evaluation

Obviously, our service providing server and object managing server can tolerate single-point failures. This is clearly an advantage compared with the normal single server model. However, every fault-tolerant method employs redundancy, and redundancy may imply increasing costs and/or decreasing performance. Similarly, protocol conversion also involves system overhead. This section tries to evaluate the performance of our server types in various circumstances.

The performance of an RPC in the SRPC system varies, according to which server type is used. Table 1 lists the null RPC performance on a network of HP and Sun workstations, where the server program runs on an HP 715/33 workstation and the server buddy and the client run on two separate Sun 4/75 ELC (33 MHz) workstations. The network is virtually isolated and very lightly loaded (no other user programs were run except the testing programs during data collection). The server (and the buddy, of course) uses the Internet datagram (UDP) protocol and the client uses the Internet stream (TCP) protocol.

The table also includes null RPC times in the SRPC system for simple servers without protocol conversion. That is, when both the client and server use the same protocol (UDP or TCP), and therefore, no protocol conversion is required.

We can draw the following observations and explanations from Table 1:

- The overhead of protocol conversion is light. From the table we know the average simple RPC time using UDP and TCP protocol only is about 2.59 ms. The simple RPC protocol conversion uses about 3.22 ms, which is only 0.63 ms more than the time used by simple RPCs without protocol conversion. When
a network is normally loaded, a null RPC time typically needs 5–10 ms. In that case, the extra time for protocol conversion could be less than 10% of the RPC time.

The overhead of using service providing server is minimum. From the table we can see that the time difference between a simple RPC and a service providing RPC is only 0.15 ms. Most of the extra time is spent on the preparation of using the buddy server in case of server failure.

The overhead of using object managing server is quite high. This is because of the server RPC used in keeping the consistency between the two objects managed by the server and the buddy. However, the time used is less than the time of two simple RPCs. This is because that there is no protocol conversions between the server and its buddy, and some of the operations are carried out in parallel.

We are still investigating the system performance under other circumstances, such as RPCs with various sets of parameters and with various network load conditions.

**Stub and driver generator**

1. **Syntax**

The purpose of the stub and driver program generator is to generate stubs and driver programs for server and client programs according to the Server Definition Files (SDF). The syntax of a server definition file is shown in Figure 1.

We use a modified BNF to denote the syntax of finction files. The ‘variable’, ‘integer’, ‘string’, ‘constant’, and ‘declarator’ have the same meanings as in the C programming language. Comments are allowed the definition file. They are defined the same as in the programming language using /* and */.

2. **Semantics**

Most of the descriptions of Listing 1 are self-explanatory. We only highlight the following points:

- The server’s name is defined as a variable in the C language. This name will be used in many places.

```
<HEADER>: Server Name:
variable;
Comment: string;
Server Protocol:
variable;
Client Protocol:
variable;
[

<BUDDY>:
Buddy <BDTYPED>: variable;
Using: <LANGUAGE>;
<BDTYPED> := Auto | Forced
<LANGUAGE> := C | Pascal
<CONST> := constant
<FUNCTIONS> := RPC Functions:
<RPCS> := <RPC> { <RPC> }
<RPC> := Name: string
[Update]; <PARAMS>
<PARAMS> := 
<PARAM> := Param: <CLASS>:
declarator;
<CLASS> := in | out
```

Listing 1. Server definition file syntax.

For example, it is the key in the LS database to store and access server entities. When the client asks the LS to locate a server, it provides the server’s name defined here. The name is also used as a prefix in naming all the files generated by the SDG. So two different servers cannot be assigned to the same server name. Otherwise, the server who registers to the LS first will be accepted while the server who registers to the LS later will be rejected by the LS.

2. Different protocols can be defined for the server and the client, respectively. The buddy, if it is defined, uses the same protocol as the server does. Currently, only three protocols are allowed: Internet_datagram (The UDP protocol), Internet_stream (the TCP protocol), and XNS_datagram (the XNS packet exchange protocol).

3. The <BUDDY> part is optional. If it is not specified, the generated server will be a simple server, otherwise it will be a service providing server or an object managing server, according to some definitions in the <RPCS> part (described below). The <BUDDY> part has a buddy definition and a language definition. The buddy definition defines that whether the buddy’s name and execution is to be determined by the system
(Auto) or to be determined by the programmer (Forced). If Auto is defined, the system will generate the buddy server’s name (Server Name) used for registering and locating it), the buddy’s driver and stub files as well as the makefile, and will treat the following variable as the name of the buddy’s procedure file. Then, the buddy program will be compiled and executed together with the server program. The host of the buddy program will be determined by the system at run time.

If Forced is defined, the generator will not generate any buddy’s program file and will treat the following variable as the name of the buddy server used for registering and locating. The programming and execution of the buddy server will also be the programmer’s responsibility.

The language definition Using defines which language does the buddy program use. The key issue of software fault-tolerant is the diversity of version independent, and one way of achieving design diversity is through the use of multiple programming languages [24]. If a different language is chosen for each version implementation, then the versions are likely to be more independent, not only due to the diversity of languages, but also because individual language features force programmers toward different implementation decisions. Currently only the C programming language is supported in the SRPC system. We have planned to support the Pascal language implementation soon.

4. The <FUNCS> part defines the remote procedures of the server. At least one remote procedure must be defined. Each remote procedure is defined as a name part and a parameter (<PARAMS>) part. The name of a remote procedure is simply a variable, with an optional Update definition. The latter definition distinguishes an object managing server with a service providing server. That is, if the <BUDDY> part is defined and the Update is defined in any one RPC definition, the server is an object managing server. If only the <BUDDY> part is defined but no Update part is defined in any RPC definition, the server is a service providing server. The meaning of the Update definition is: if an Update is defined following an RPC procedure name, that procedure must be maintained as a nested RPC affecting both the server and the buddy by the server program (see Section 3.1).

There can be zero or several parameters in a procedure, each consisting of a class and a declaration. The class can be in or out, which tells the SRPC system that the parameter is used for input or output, respectively. The declaration part is the same as in the C language. In this version, only simple character string is allowed in parameter definitions. Further extensions are under way.

4.3. Implementation issues

After a programmer sends a server definition file to the generator, the generator first does syntax checking. If no errors are found, several program source files and a makefile are generated. The subsequent processing is specified by the makefile. That is, when using the make utility, the executable files of both the server and client will be generated. Fig 4 indicates the structure of the processing. The dashed lines represent optional actions.

At least one server definition file must be input to the SDG. If there are more than one server, their SDFs can be input to the SDG simultaneously. If there is only one SDF file, then the generated client driver can execute the server’s procedures one by one. If the buddy part is also specified, the generated client can also call the buddy procedures directly (this is useful in testing the client-buddy communication).

If there are more than one SDF file, then for each server, the SDG will generate one set of server files, one set of client files, and one set of buddy files (if the buddy is defined), respectively. These files are the same as the servers being processed in single file input described above. One additional set of client files, the mult-server client program, will also be generated in this case. The client driver is called a mult-server client driver. It can call all the procedures of all the servers one by one. A further improvement is under way to let the client call these procedures in parallel.

5. Application example

We use a simple example to show the application of the SRPC system. Suppose we have a server definition file called sdef. It defines a ‘send-and-forward’ system in that the server acts as a message storage and the client acts as both message sender and receiver. The server definition file is shown in Listing 2.

From the header part of this SDF file we know the following: the server is named as s and the server protocol used is the Internet datagram. The server can
two RPC function definitions have no Update marks, and then they will be treated as ordinary RPCs.

When the file is input to the generator, the following files will be generated:

- sf/Header file, must be included by server, its buddy and client drivers and stubs.
- sfSer.c: Server driver file.
- sfOps.c: Frameworks of server procedures.
- sfCl.c: Client driver file.
- sfStubCl.c: Client stub file.
- sfBdy.c: Server buddy driver file.
- sfStubBdy.c: Server buddy stub file.
- makefile: Makefile.

After using the make utility (simply use 'make' command), three executable files are created:

- sfSer: Server program.
- sfCl: Client program.
- sfBdy: Server buddy program.

Note that the sfOps.c file only defines the frameworks of the remote procedures (dummy procedures). Their details are to be programmed by the programmer. The sfBdyOps.c file should be the same as the sfOps.c file (the only possible difference happens when the server buddy uses another programming language such as the Pascal; then the affix of the file would be .pas).

The server driver is simple. It does the initialization first, then it registers with the LS and invokes the buddy program on a neighbouring host because the buddy is defined as Auto in the SDF file. After that it loops 'forever' to process incoming calls until the client issues a 'shutdown' call. In that case, the server un-registers from the LS and exits. The 'un-register' call will automatically un-register the buddy from the LS as well. The incoming calls are handled by the server stub and underlying library functions. A listing of the server driver is shown in Listing 3.

The server buddy driver works in the same way as the server program, except that it does not invoke a buddy program. Also, the buddy is a simple server and all calls to the buddy will not be nested.

The generated client driver can execute the server's remote procedures one by one. If the server driver is running and the client driver is invoked, the client driver first lists all the remote procedures provided by the server, and asks the user to choose from the list. The following is the menu displayed for this example:

Listing 2. Server definition file example.
Initialisation (including invoke the buddy);
/* Register the server to the LS */
registerServer("sf", "sfBdy", imp);
while (1) {
    wait for client calls;
    /* comes here only if a client called */
    fork a child process to handle the RPC;
    if the call is "shutdown"
        break;
}
unregisterServer("sf");

Available calls:
0 sf$Shutdown
1 sf$storeMsg(receiver, msg, stat)
2 sf$forwardMsg(receiver, msg)
3 sf$readMsg(receiver, msg)
4 sf$listMsg()

Your choice:

After the selection, the input parameters of the named remote procedure are then input from the keyboard. After that, the driver program does some initialization and the remote procedure is executed and returned results displayed. The actual calling and displaying are handled by the client stub and underlying library functions. The format of all the four RPCs in the client program are the same as the format listed in the above menu. That is, if the client wants to send a message to a receiver, it does the following call after the receiver's name and the message are input into receiver and msg variables, respectively:

sf$storeMsg(receiver, msg, stat);

Note that the remote procedure's name is named as a composition of the server's name sf, a $ sign, and the remote procedure's name storeMsg in the SDF file. Similarly, if the client wants to receive messages, it does the following call after the receiver's name receiver is obtained:

sf$forwardMsg(receiver, msg);

Before each RPC, a locateServer("sn", buddy, imp) call is issued to the LS to return the location of the server and the name of its buddy. The server location is stored in imp and the buddy name is stored in buddy.

The fault-tolerant feature of the system is completely hidden from the user. For this example, all the remote procedure calls from the client program will be first handled by the server. A nested RPC is issued if the incoming call is either sf$storeMsg(receiver,
msg, stat) or sf$forwardMsg(receiver, msg). This is because the two RPC functions are marked as Update in the SDF file. The nested RPC will ensure that actions of the incoming call will be made permanent on both the server and its buddy, the call is successful, and no actions of the incoming call will be performed if the call fails. Two other incoming calls, sf$readMsg(receiver, msg) and sf$listMsg(), will be handled by the server only.

If the server fails (that is, the RPC to the server returns an error), the client program will send the RPC to the server's buddy. The location of the buddy will be determined by another call to the LS:

locateServer(buddy, ",", imp)

where buddy is the server buddy's name obtained during the first call to the LS, and imp stores the location of the buddy.

The cross-protocol communication is also hidden from the user. All the interfaces to the protocol converters (client-LS, client-server, and server-LS) are generated by the SDG (in the stub files) and used automatically by the stubs. If a user only deals with the RPC level, he or she will never notice the underlying protocols used by the server and the client programs.

The termination of the server program also needs to be mentioned. After the server program is started, it will run forever unless the programmer kills its process, or there is a facility to terminate the server. Here we provide a facility to do that job. We add a 'remote shutdown procedure into the server, and allow the remote shutdown of the server in the server program. Hence when the client driver calls the remote shutdown procedure of the server, the server will shut down itself and will exit from the system.

6. Remarks

A system for supporting fault-tolerant, object-oriented software development is described in this paper. The system is simple, easy to understand and use, and has the ability of accommodating multiple communication protocols and tolerating server failures. It also has the advantage of producing server and client driver programs, and finally executable programs directly from the server definition files. The system has been used as a tool of distributed computing in both third year and graduate level teaching, and has been used by some students in their projects.

In tolerating server failures, similar efforts can be found in the RPC systems that provide replicated server facilities, such as NCA/RPC [13]. But in these systems, the user, instead of the system, takes the responsibility of maintaining and programming the functions for object consistency. This is a difficult job for many programmers.
he Argus system and other systems that maintain transaction atomicity also provide some sort of fault tolerance for servers (guardians in the Argus), but their purpose is to maintain the transaction atomicity, that is, if a server fails the transaction may abort and it has no effect on the committed objects, and other transactions will not be effected. Our approach in achieving fault tolerance is similar to the approach used in the ISIS toolkit (of course, ours is more simplified and less powerful). But our system is simple, easy to understand and easy to use. In our system, we provide a server buddy to tolerate the server's failure. When the server fails, the client, instead of aborting, can access the server buddy to obtain the tentative service. Also in our system, it is the system, and not the user, that is responsible of maintaining the consistency of the managed objects.

Providing server and driver programs directly from the server definition file (similar to the interface definition es of other RPC systems) is also an interesting characteristic of our system. It is related to the rapid prototyping of RPC programs [25]. The driver programs are simple, but have the advantage of testing the executability of the RPC program immediately after designing the SDF file. It is especially useful if the user makes some changes in the SDF file or the procedure file. In that case, these changes will be automatically incorporated into other related programs if the program is re-generated by the stub and driver generator. This will avoid a lot of trouble in the maintenance of consistency of program files.

References


15. B. Liskov, Distributed programming in ARGUS, Comm. ACM, 31(3) (March 1988) 300–312.


Wendel Zhou received the BEng and MEng degrees from Harbin Institute of Technology, Harbin, China in 1982 and 1984, respectively, and the PhD degree from the Australian National University, Canberra, Australia in 1991. He is currently a Lecturer in the School of Computing and Mathematics, Deakin University, Geelong, Australia. Before joining Deakin University, Dr Zhou was a programmer in Apollo/HP at Massachusetts, USA, a lecturer at the National University of Singapore, Singapore, and a lecturer at Monash University, Melbourne, Australia. His research interests include distributed computing, computer networks, performance evaluation and fault-tolerant computing.
Fault Detection and Fault Tolerance in a Loosely Integrated Heterogeneous Database System

Wanlei Zhou

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

This paper presents the design and prototype implementation of a reactive system that facilitates the fault detection and fault tolerance for a loosely integrated heterogeneous database system. The fault detection mechanism uses sensors to monitor individual databases and system objects and to detect database or system component failures. The fault tolerance mechanism uses actuators to react on these failures.

1 Introduction

A distributed database system consists of many hardware and software components that are likely to fail eventually. In many cases, such failures may have disastrous results. It is therefore essential to build distributed database systems that can tolerate component failures.

However, the development of fault-tolerant software is a very difficult task. One reason is that, in normal practice, fault-tolerant computing policies and mechanisms are deeply embedded into most application programs. Therefore these application programs cannot cope with changes in environments, policies, and mechanisms very well. To build fault-tolerant computing systems that can cope with the constant changes in environment and user requirements, it is essential to separate the fault-tolerant computing policies and mechanisms in application programs.

The reactive system concepts are an attractive paradigm for system design, development, and maintenance because it separates policy from mechanism. Several systems, such as Meta\(^1\), DisCo\(^2\) and STATEMENT\(^3\), and languages, such as Reactive C\(^4\) and Reactive Pascal\(^5\) that based on the reactive system concepts have been developed recently.

However, most of the research on reactive systems are concentrated on process control (such as controlling a robot). This paper applies the reactive system concepts in developing a fault-tolerant heterogeneous database system.

The paper is organised as follows. Section 2 briefly introduces the architecture of the loosely integrated heterogeneous database system and the models used in design fault-tolerant servers in a distributed system. Section 3

\(\text{(0-7803-4229-1/97$10.00 \quad 1997 \text{ IEEE})}\)
introduces our reactive system model. Section 4 presents the design and implementation issues. Section 5 describes the fault detection and fault tolerance, and finally, Section 6 summaries the paper.

2 The Loosely Integrated Database and Fault-Tolerant Models

This section describes the loosely integrated heterogeneous database system that we use in our case study and the fault-tolerant models that we can apply to tolerate component failures of our database system.

2.1 The Loosely Integrated Heterogeneous Database System

Most of the existing approaches in integrating heterogeneous database systems use passive information sharing. That is, they let some authority (e.g., the global schema) decide what are the content and format of information sharing. We have implemented a loosely integrated heterogeneous database system by using active information sharing. That is, existing database systems are allowed to maintain their autonomy, yet through their willingness, provides a substantial degree of information sharing. The participating database systems themselves decide the content and format of information sharing.

![Diagram](image)

Figure 1: Design architecture of the loosely integrated system

Figure 1 depicts the architecture of our loosely integrated heterogeneous database system. Currently two database systems (Oracle and Ingres) are running on two separate Sun workstations. A third database server, the Ingres database system, is to be added into the system soon.

The system consists of the following components:
• Database servers. Each individual database server maintains database operations directed to it.

• The trader. We use a trader to manage the shared information among participating databases of the loosely integrated database system. If a database system is willing to share part of its information with other database systems, it exports that information as a service offer to the trader. Application programs (clients) wishing to make use of the shared information have to import such service offers from the trader and then access the database(s) concerned.

• The trading manager. A trading manager is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results.

• Trading agents. Trading agents are appointed by database servers to manage some special service offers. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then a trading agent can be appointed to manage the offer. A trading agent exports the service offer it manages to the trader. The client imports the service offer from the trader and is given the agent’s address (and the description of the service). It then calls the agent and obtains the service.

The trader, trading agents, and user application programs can be executed on any Sun workstations. Currently the two database servers are running on two separate Sun workstations located about 100 km apart. To improve the reliability of the system, an instance of the trader is running on each location. Figure 2 shows the implementation of the system.

2.2 Fault-Tolerant Models for Servers

In and we have proposed four types of fault-tolerant servers, two of them are described below.

A service providing server (Figure 3(a)) has a buddy server running (or in a cold stand-by state, i.e., ready to be executed) somewhere in the network, but no communication between the server and its buddy. When a service providing server fails, an operation to this server will be automatically re-directed to its buddy server by the system (the buddy server will be activated if it is in the cold stand-by state).
A twin-server (Figure 3(b)) has two servers running in parallel to provide the fault-tolerant service. We call them as twins and denote them as \( T^1, T^2 \). These two servers usually (but not necessarily) locate on different hosts for increasing reliability.

When a client (e.g., Client 1) requests a service, the request is mapped to one of the twins (e.g., \( T_1 \), the solid line) for processing. At the same time, a copy of the request is also sent to \( T^2 \) and is kept in its reserve (the dashed line). If \( T^1 \) has completed the service of the request, it notifies \( T^2 \) and \( T^2 \) will drop the copy of the request from its reserve.

At any time, if one of the twins, say \( T^1 \), dies for some reason, the alive twin \( T^2 \) will continue the service of \( T^1 \) by using the requests kept in its reserve.
Then $T^2$ will invoke another server to be its new twin (also named as $T^1$). After that, the normal operation of the twins is resumed.

In our heterogeneous database system, the priorities of fault-tolerance considerations for different servers can be listed, from high to low, as follows: trader, database server, trading manager, and trading agent. The trader is then implemented as a twin server and others are implemented as service providing servers.

3 The Reactive System Model

Figure 4 depicts the architecture of our reactive system model. In this model, sensors are attached to applications to obtain their states (or equivalently, to monitor some events about the applications). These events are sent to the decision making managers (DMMs). The DMM reacts to these events by using the actuators to change the states of the applications. The model consists of the following levels:

- **Fault-tolerant computing applications.** This level deals with application-specific issues.

- **Fault-tolerant computing mechanisms.** This level deals with all mechanisms for implementing the fault-tolerant computing strategies. For example, it deals with mechanisms used in detecting and reporting component failures, and mechanisms used in masking and recovering from component failures.

- **Fault-tolerant computing policies.** This level deals with policies in fault-tolerant computing. These policies are implemented through decision making managers. A DDM obtains information about applications via
sensors, processes the information, and modifies the states of applications via actuators in order to tolerate component failures.

Sensors and actuators can be simple or composite. A simple sensor (simple actuator) can only be directly attached to one (local) application. Figure 5(a) shows a simple sensor (actuator). A composite sensor (composite actuator) can be composed of multiple sensors (actuators) from multiple applications (Figure 5(b)).

Figure 5: Sensors and actuators

4 Design and Implementation Issues

The main responsibility of sensors is to monitor events. In order to monitor events we have to assign each event a unique name. The following definitions are a simplified version based on 19.

**Definition 1.** An event name is a text object with the following syntax: event.name := EventID.HostID, where EventID is a unique number (within a local computer), and HostID is an identifier for the host on which the event occurred.

**Definition 2.** A simple event is defined as a pair (en, m) where en is an event name and m a message. In this context a message is some information associated with the event. For example, we can associate the name of the event that causes this particular event, a time-stamp, or a programmer-defined string used to characterise one or more event attributes with the message part of an event.

**Definition 3.** Let e, e₁ and e₂ be events. The following operations are defined over events:
• Happens. We use $e$ to mean that event $e$ happens.

• Not happen. We use $\neg e$ to mean that event $e$ does not happen within a preset time period. $\neg e$ is still regarded as a simple event.

• Or. We use $e_1 \lor e_2$ to represent the fact that either $e_1$ or $e_2$ or both of them happen.

• And. We use $e_1 \land e_2$ to represent the fact that both $e_1$ and $e_2$ happen.

Composite events are made from simple and/or composite events by applying event operations.

In our design, a simple sensor is responsible for monitoring one simple event. Sometimes it is convenient to apply the above event operations to simple sensors. In that case we actually mean that the operations are applied to the simple events that are monitored by these sensors. According to the way that a sensor performs its work, we have defined the following three types of simple sensors:

• Event sensor. It reports to its subscribers when the monitored event happens.

• Timer sensor. It periodically sends out some message to its subscribers.

• Polling sensor. It periodically checks some state of the application and reports to subscribers.

A local event manager (LEM) runs on each host and manages communication activities (such as event reporting) for subscribers and sensors located on remote machines. If an event happens, the event is reported to local and remote subscribers through different methods (Figure 6):

• Local. Simple sensors can only report to local subscribers directly.

• Remote. Remote event reporting is managed by the local event manager that runs on the local machine.

In Figure 6, when the event monitored by sensor $S_i$ happens, the event is reported directly to all the local subscribers, including the local event manager. The LEM is then responsible for propagating the event to all the remote subscribers. The main reason for separating the processing of local and remote
event reporting is to make sure that the local subscribers get the sensor's event reporting as fast as possible.

In our design, the DMM is where the application policies are specified and maintained. It uses actuators (procedures that change the state of applications) to perform the required work on applications. For example, an actuator can perform the task of simply setting a value, or the task of activating a buddy server.

It is obvious that some form of one-to-many communication facility is required in implementing the system. We use UNIX stream sockets to build local one-to-many communication primitives and the Internet stream sockets to build remote one-to-many communication primitives.

The following one-to-many communication primitives are used in our implementation:

- **groupCreateLocal().** It creates a local one-to-many communication group where one object acts as the sender and others the receivers.

- **groupDestroy().** It destroys a communication group.

- **groupJoinLocal().** An object uses this primitive to join a local one-to-many communication group as a receiver.

- **groupJoinRemote().** An object uses this primitive to join a remote one-to-many communication group as a receiver.

- **groupLeave().** An object uses this primitive to leave a one-to-many communication group (the object must be a receiver of the group).
• **groupSendLocal()**. A sender uses this primitive to send a message to all the local receivers of the group.

• **groupSendRemote()**. A sender uses this primitive to send a message to all the remote receivers of the group.

A simple sensor uses the following function (called the *event report function*) to report an event:

```
sensor_report_event(localSubscribers, event)
```

where `localSubscribers` contains the local subscribers to this sensor (it is actually a *group ID*). Local subscribers can be the local DMM, the local event manager, or a local sensor; and the `event` is a simple event (*en, m*). The function uses the communication primitive `groupSendLocal()` to send an event report to all local subscribers. The LEM is then responsible for sending the event report to all remote subscribers using the `groupSendRemote()` primitive.

Simple sensors must be closely attached to their applications. We use the following two methods to create simple sensors according to the types of sensors:

• **Event and timer sensors.** These sensors are created by inserting into an application program the event report function. This implies that the source code of the application program must be available and the recompilation of the application program is necessary. Most of the servers (the trader, the trading managers, and the trading agents) can use these sensors.

• **Polling sensors.** These sensors are created as independent processes that run in parallel with their applications. This implies that polling sensors can be used to monitor applications that we do not have their source code (such as the Oracle database server).

The communication primitive `groupCreateLocal()` is used to create the initial one-to-many communication group of each sensor. The communication primitive `groupJoinLocal()` is used to subscribe to a sensor locally and the primitive `groupJoinRemote()` is used to subscribe to a sensor remotely.

The facility for creating a composite sensor has been planned but has not been implemented yet. The principle of creating a composite sensor is to use composite events.

5 Fault Detection and Fault Tolerance

Different types of sensors are used to detect faults in different servers of the system.
• The trader. The trader is implemented as a twin-server. That is, two
trader servers are executed on different computers to provide the service.
Event and timer sensors are placed into various positions of the trader
(Figure 7).

![Diagram of the trader](image)

**Figure 7: Fault detection: the trader**

We use $T^1.S_j$ to denote the sensor $S_j$ attached to the trader server $T^1$.
Since $T^1$ and $T^2$ are symmetric, we only describe the functions of sensors
for $T^1$.

The timer sensor $T^1.S_1$ periodically reports to subscribers the liveness
of trader server $T^1$. So,

$$\text{if } \neg T^1.S_1, \text{ then } T^1 \text{ is faulty.} \quad (1)$$

The time interval for evaluating this is set to be greater than the time
interval of $T^1.S_1$.

The event sensors $T^1.S_2, T^1.S_3, T^2.S_2$, and $T^2.S_3$ monitor the inter-
communication between the twins. $T^1.S_2$ monitors the event where a
request for a nested operation from $T^1$ to $T^2$ takes place. $T^2.S_3$ monitors
the event where a returning of a successfully executed nested operation
by $T^2$ on behalf of $T^1$ takes place. Therefore,

$$\text{if } T^1.S_2 \land \neg T^2.S_3 \text{ then the nested operation fails.} \quad (2)$$

The combination of event sensors $T^1.S_4$ and $T^2.S_5$ provides the possi-
bility of consistency checks for the normal queue of $T^1$ and the reserve
queue of $T^2$. $T^1.S_4$ monitors the event where an input to $T^1$'s normal
queue takes place, while $T^2.S_5$ monitors the event where an input to $T^2$'s
reserve queue takes place. Therefore,

$$\text{if } T^1.S_4 \land \neg T^2.S_5 \text{ then the queues are not consistent.} \quad (3)$$
The database servers. A polling sensor is attached to each database server to periodically check the liveness of the database server (Figure 8(a)). So,

\[
\text{if } \neg S, \text{ then the DB server is faulty.} \tag{4}
\]

The time interval for evaluating this is set to be greater than the polling time interval of $S$.

Trading managers and trading agent. A timer sensor is inserted into each trading manager and each trading agent to report periodically to subscribers (Figure 8(b), (c)). So,

\[
\text{if } \neg S, \text{ then the trading manager/agent is faulty.} \tag{5}
\]

The time interval for evaluating this is set to be greater than the time interval of $S$.

![Diagram](image)

Figure 8: Fault detection: database servers, trading managers, and trading agents

Various fault tolerance policies are implemented by DMMs through the help of actuators.

On the host that the trader server $T^1$ is running, there is a DMM ($DDM_{T^1}$) that subscribes to $T^1.S_1, T^1.S_2$, and $T^1.S_4$ locally and to $T^2.S_1, T^2.S_2$ and $T^3.S_5$ remotely (since the two twin-servers running on different hosts). Similarly, on the host that $T^2$ is running, a DMM ($DDM_{T^2}$) subscribes to $T^2.S_1, T^2.S_2$ and $T^2.S_4$ locally and to $T^1.S_1, T^1.S_3$ and $T^1.S_5$ remotely.

Both $DDM_{T^1}$ and $DDM_{T^2}$ use formula (1) to monitor the liveness of $T^1$. However, $DDM_{T^2}$ takes the initiative to make the decision on what to do after knowing that formula (1) is true. In that case, $DDM_{T^2}$ asks $DDM_{T^1}$ to confirm that its evaluation of (1) is also true. If so, it asks $DDM_{T^1}$ to restart the twin server on the original computer. If $DDM_{T^1}$'s evaluation of formula (1) is false, then a second evaluation by $DDM_{T^2}$ is carried out. If the second time the two DMMs still disagree with their evaluation, then the twin-server
is restarted on another computer. DDM\textsubscript{T\textsuperscript{2}} also moves all jobs in T\textsuperscript{2}'s reserve queue to T\textsuperscript{2}'s normal queue.

Listing 1 shows the algorithm `liveness_checking\_DMM2_to_T1()` used by DDM\textsubscript{T1} for checking the liveness of T\textsuperscript{1}. The algorithm used by DDM\textsubscript{T2} for checking the liveness of T\textsuperscript{2} is symmetric.

```
Listing 1: Algorithm used by DDM\textsubscript{T1} for checking T\textsuperscript{1}

liveness_checking\_DMM2_to_T1()
{
  int checkFlag = 0;
  while (true) {
    /* receive report from T\textsuperscript{1}.S\textsubscript{1} */
    /* evaluate the event reported by T\textsuperscript{1}.S\textsubscript{1} */
    if evaluated by DDM\textsubscript{T1}: \neg T\textsuperscript{1}.S\textsubscript{1} = true then
      ask DDM\textsubscript{T1} to evaluate: \neg T\textsuperscript{1}.S\textsubscript{1};
    if evaluated by DDM\textsubscript{T1}: \neg T\textsuperscript{1}.S\textsubscript{1} = true then
      /* now both DMMs agree with each other that T\textsuperscript{1} fails */
      ask DDM\textsubscript{T1} to restart the twin-server locally;
      moves all jobs in T\textsuperscript{2}'s reserve queue to T\textsuperscript{2}'s normal queue;
      checkFlag = 0; /* reset the flag */
      continue; /* go back to while */
    endif;
    /* comes here if both DMMs do not agree with each other */
    if checkFlag == 0 then /* first time */
      checkFlag = 1; /* set the flag */
      continue; /* go back to while */
    else /* second time, and both DMMs do not agree with each other */
      restart the twin-server on a different computer;
      moves all jobs in T\textsuperscript{2}'s reserve queue to T\textsuperscript{2}'s normal queue;
      checkFlag = 0; /* reset the flag */
      continue; /* go back to while */
    endif;
  } /* while */
}
```

If DDM\textsubscript{T1} has evaluated that formula (1) is true, then it sets a timer to wait for the initiative action from DDM\textsubscript{T2} and responds accordingly (as expected in `liveness_checking\_DMM2_to_T1()`). If DDM\textsubscript{T2} does not contact DDM\textsubscript{T1} within the time limit, DDM\textsubscript{T1} will start the failed server on the
original (local) computer.

Checking the failures of nested operations between $T^1$ and $T^2$ are also the responsibility of the two DMMs. For example, $DMM_{T^1}$ subscribes to $T^1.S_2$ and $T^2.S_3$ in order to monitor the nested operations from $T^1$ to $T^2$. It uses formula (2) to decide if a nested operation from $T^1$ to $T^2$ fails. That is, if $T^1.S_2 \land \neg T^2.S_3$ is true, then a nested operation has been sent out from $T^1$ to $T^2$ but a successfully response has not been issued. In that case, $DMM_{T^1}$ will use an actuator to re-start the nested operation.

The two DMMs are also responsible for checking the consistency of the queues on the two twin-servers. For example, $DMM_{T^1}$ subscribes to $T^1.S_4$ and $T^2.S_5$ to monitor the new requests going into the normal queue of $T^1$ and the reserve queue of $T^2$, respectively. If a request enters the normal queue of $T^1$ but not the reserve queue of $T^2$ (that is, formula (3) is true), then $DMM_{T^1}$ uses an actuator to re-send the request to $T^2$'s reserve queue.

A separate DMM runs on each machine that has a database server, or a trading manager, or a trading agent running on it. The responsibility of the DMM is to monitor the liveness of these servers running on the local machine, by subscribing to the sensors monitoring these servers. If such a server fails (by evaluating formulas (4) and (5)), the DMM will then use an actuator to restart the server.

The policies for dealing with a database server failure and for dealing with a trading manager/agent failure is a little different. Currently, two database servers are running on two different machines. Let us denote the machine that runs Oracle as $M_o$ and the machine that runs MinisQL as $M_m$. We also have a cold stand-by Oracle server on $M_m$ and a cold stand-by MinisQL server on $M_o$. The DMM on $M_o$ monitors the liveness of the MinisQL database server using a similar algorithm as in Listing 1. The same algorithm is also used by the DMM on $M_m$ to monitor the liveness of the Oracle database server. The purpose of this is to make sure that the database servers can be restarted even after a host failure.

When a DMM detects a failure of a local trading manager/agent, it then uses an actuator to restart the failed trading manager/agent.

6 Summary

In this paper we have shown the fault detection and fault tolerance in a loosely integrated heterogeneous database system through the use of the reactive system model. By using the reactive system model, we have successfully separated fault detection and fault tolerance mechanisms from their policies. This separation allows us to apply various fault-tolerant policies that use various fault
detection mechanisms.

References

A tool for layered analysing and understanding of distributed programs

Wanlei Zhou

School of Computing and Mathematics, Deakin University, Geelong, Victoria 3217, Australia

Received 30 June 1995; revised 25 September 1996; accepted 11 October 1996

Abstract

Distributed programs are harder to analyse and understand than sequential programs for a number of reasons. First, a distributed program consists of many entities at different levels of abstraction. Second, communication among the entities of a distributed program may happen concurrently and nondeterministically. To be able to analyse these entities and the communication events among them is therefore essential for understanding any distributed program. This paper describes a tool for assisting the analysis and understanding of distributed programs. The tool analyses the entities and communication events of a distributed program at four levels of abstraction. At the highest level, the tool displays the communication relationships between programs (e.g. server and client programs). At the second level, the communication between processes is displayed. At the third level, the communication between events is displayed. The fourth level is the lowest: it uses a text editor to show the relevant statements that carry out the communication. The tool has been used in the teaching of courses related to distributed computing since 1993. © 1997 Elsevier Science B.V.

Keywords: Distributed systems; Event-based debugging; Execution replay; Program analysis

1. Introduction

Distributed computing has now become the norm in computing. However, distributed programs are much more difficult to analyse and understand than sequential programs [6,20], due mainly to the parallelism and asynchrony inherent in a distributed program.

A distributed program consists of many entities at various levels. At the highest level, a distributed program can be viewed as a group of program parts (each of which can be an independently compiled program, e.g. the client and server program parts) that work together on a single task. Each program part can be further split into several processes during the program's execution. Each process consists of a sequence of events that may happen during the execution. The concurrency and communication among these program entities are the main reasons that make analysing and understanding of distributed programs so difficult [5]. To be able to analyse these entities and the concurrency and communication among them is therefore essential for understanding any distributed program.

One technique, called debugging with trace of program execution [10,17] has long been used in debugging parallel and distributed programs. The essence of this technique is to map the non-deterministic concurrency and communication among entities of the program into one of the deterministic cases for a further study. This technique is also suitable for analysing and understanding a distributed program, and has two phases. The first phase generates a trace of the program's execution. The generated trace is no longer non-deterministic and can be analysed in any controlled way preferred by the programmer. The second phase is to analyse the trace by replaying the distributed execution [9].

Generating an execution trace of a distributed program can be very costly in time and space. In the behavioural abstraction approach proposed by Bates and Widen [3], a distributed program is a stream of events that can be filtered and clustered so that unnecessary events can be ignored and groups of events can be hierarchically defined into higher-level events in terms of sequences of primitive events (such as process creation, page fault, and message exchanges). In his later work [1,2], Bates described a technique called event-based models of behaviour (EBBA) for debugging on heterogeneous distributed systems based on his behaviour abstraction. The EBBA technique can enhance debugging-tool transparency, reduce latency and uncertainty for fundamental debugging activities, and accommodate diverse, heterogeneous architectures. Lazzerini and Prete [14] introduced the concept of a compound event, which is expressed in terms of either an accumulated event, a sequential event conjunction, a logical event disjunction or an instantaneous event conjunction. Cheng and Wallentine [4] used a knowledge-based language to specify the events in distributed debugging. In their language, an
aggregate event can be constructed from primitive events and previously defined aggregate events. In Ref. [9], the authors aim at reducing the amount of data one has to record during an execution by ignoring communication events that are not involved in message races. Two messages are said to be in a race condition if their destinations are the same and their order of arrival at the destination is determined by arbitrary conditions only (such as transmission over the network). The debugger for Amoeba [7] was also modeled after the behavioural abstraction approach.

Replay is a technique that allows a user to examine the course of an erroneous execution without re-executing the program. LeBlanc and Robbins [15] provided some degree of replay in their debugger. After the collection of all events, the events are displayed sequentially. Both single step and continuous display are supported. Leu et al. [16] proposed an efficient execution replay technique that improved LeBlanc’s algorithm in the case of distributed programs. Hurfin et al. [12] described a replay system that operates at the source level; they used Estelle, an ISO normalised high-level language for protocol specification and prototyping.

All of these replay techniques suffer from one drawback, however, when they are used in analysing and understanding distributed programs. They display events in a flat fashion and in great detail because they all assume that the person who uses the debugger is already familiar with the structure (or was the programmer) of the distributed program. Unfortunately, in the case of analysing and understanding a distributed program, the person usually has little knowledge of the program and will be lost in a mountain of detail.

Many efforts have been made to visualise distributed and parallel program execution to facilitate the analysing and understanding of these programs [21,19]. However, most of these tools are complex and are therefore not useful for novices, and usually do not allow easy study and experimentation with programs. Some existing tools provide graphical pictures of algorithms or data structures in action [18,11], but do not display the structures and source code of programs.

This paper describes a tool for assisting in the analysing and understanding of distributed programs. It is a significant improvement on our earlier distributed monitor [23,24] and has been used in teaching of courses related to distributed systems since 1993 (interested readers can obtain the source code of the tool by contacting the author). The tool helps a user to understand a distributed program in a top-down fashion, as indicated in Fig. 1.

At the top level (program level), the tool displays the communication relationships between program parts (e.g. server and client programs). The top level gives the user the overall function of all program parts involved in the distributed computing. At the second level (process level), the communication between processes is displayed. Because a program part can split into several processes, which can run concurrently with processes from other program parts, the second level therefore gives the user a clear picture of the cooperation among all (or a selected group of) processes. At the third level (event level), events related to some particular communication and a partial ordering among these events are displayed. This level gives the user a closed look on the communications and order of events. The fourth level (source level) is the lowest; it displays the relevant statements that carry out some particular communication.

The rest of the paper is organised as follows. Section 2 presents an example of our top-down fashioned analysis and understanding of a distributed program. Section 3 briefly describes the structure of the tool, and Section 4 describes issues related to its implementation. Section 5 proposes some expansion on the tool, and Section 6 presents some conclusions.

2. An example

We use a simple example to illustrate the analysis and understanding process of a distributed program using our tool. The program is a "send-and-forward" system. The server program (named as ServerPrsg) acts like a message storage. The user uses the client program (named as ClientPrsg) to send a message (with a receiver's name) to the server. The server keeps the message until the addressed receiver uses the client program to ask the server to forward messages.

We start the server part of the program first. Then a client part is used to send a message to the server. After the program is executed under the monitoring of the tool, we have the top-level diagram of Fig. 2. From this diagram we know that the client initialises a request. The server acknowledges and then the client sends the message to the server. The server then sends back an acknowledgement to the client after receiving the message. We further use the second level to find out what processes are involved in the communication. Fig. 3 shows the process level diagram.

Three processes are involved. Process 6911 is the main process of the server program and process 6917 is the process of the client program. When the server receives a request from the client, it splits into two processes. The parent process (6911) waits for requests from other clients,
and the child process (6922) is then responsible for all the communications between the server and the client.

We can further examine the detailed communication events between processes 6917 and 6922. The third level diagram of Fig. 4 shows the event relations of these two processes. It can be seen from Fig. 4 that, after splitting from the parent process, the child process of the server program sends an acknowledgement to the client process.

The client process sends the message to the child process. Finally, the child process returns an acknowledgement to the client process. These communication events are clearly numbered in Fig. 4.

For better understanding of the communication we ask the tool to display event tables for all these three processes (Tables 1–3). Each table displays the sequential number, the event symbol, the event name, the remote predecessor, and the remote successor for all events of a process (see Section 3.1 for definitions).

Combining the level 3 diagram and the event tables can facilitate the understanding of the events associated with the processes. For example, from the event tables we know that process 6917 (client process) sends a request to the server process 6911 in event $a_4$. Then process 6911 forks a child process (6922) in event $a_9$ to manage the communication with the client. Process 6922 then acknowledges to process 6917 in event $c_5$. After that process 6917 sends the message to process 6922 in event $b_8$ and process 6922 sends back an acknowledgement to process 6917 in event $c_7$ after process 6922 receives the message.

If we want to view the detailed programming code on communication events, we can use the fourth level diagram to invoke a text editor to examine the code.

3. Structure of the tool

3.1. Definitions

Before describing the structure of the tool, we present several definitions that are important in our discussion. In order to monitor and record events we have to assign each event a name. Definition 1 is the method which can uniquely identify any event.

Definition 1. The name of an event (event name) is a “unique ID” (UID), defined as a string of characters. The format is $t.r.h$, where $t$ is the timestamp, which is stamped by the local host; $r$ is a sequential number; and $h$ is the host ID.

If $en$ is an event name, we use $en.t$ to denote the timestamp, $en.r$ to denote the sequential number, and $en.h$ to denote the host of event name $en$, respectively. $En.h$ can be used to identify on which host the event occurred, and $en.t$ denotes the time of occurrence (relative to the local host) of the event. In the case that $n$ events occurred simultaneously at the same host, $en.r$ is used to differentiate these $n$ events. It is not difficult to ensure that no $n$ ($n$ is a hexadecimal number and is currently set to $n \leq 255$) adjacent sequential numbers generated are the same. For example, we can have a sequential number generator (which is accessed sequentially by the event name generator) that generates numbers $0–255$ each time it is accessed. If the number reaches 255, it goes back to 0 and the circle begins again. So all events of a distributed program can be uniquely identified by using their event names if the maximum number of concurrent events in any host is $n \leq 255$. The upper bound of number $n$ can be easily increased if necessary.

For example, in Tables 1–3 events $a_1$ and $b_1$ have different host fields, which implies that these two sets of events belong to different hosts. However, events $a_9$ and $c_1$ have the same host field, so these two sets of events belong to the same host.

Definition 2. An event $e$ is defined as a pair $(f,m)$, where $f$ is called a fact and $m$ a message. A fact is something that
happens during the program’s execution. Note that not all facts of events are interesting to a programmer, but they may happen during the execution of the program. Facts can be, for example, the creation and destruction of a process, the issuing of a message sending call, or the issuing of a message receiving call, etc. A message is the information attached to the fact, such as the parameter values of a message sending call.

The basic relation between events is the happened before relation introduced by Lamport [13]. This relation can be easily extended to cover process creation and termination, as well as message-passing events [8]. Definition 3 defines the relationship between events in our system.

**Definition 3.** Let \( E = \{e_i\} \) be the set of all events of a distributed program. If event \( e_i \) causes the occurrence of event \( e_j \), or \( e_j \) immediately follows the occurrence of \( e_i \) within the same process, we say that \( e_i \) is a \textit{predecessor} of \( e_j \), and \( e_j \) a \textit{successor} of \( e_i \). In particular, if \( e_i \) and \( e_j \) are in different processes, we call \( e_i \) a \textit{remote predecessor} of \( e_j \) and \( e_j \) a \textit{remote successor} of \( e_i \). This is denoted as \( e_i \preceq e_j \). If \( e_i \) is a \textit{predecessor} of \( e_j \) and \( e_j \) a \textit{predecessor} of \( e_k \), then we say \( e_j \) is also a \textit{predecessor} of \( e_k \).

For example, \( e_1 \) is the event “issuing a request of sending a message” of a client program, and \( e_2 \) is the event “receiving a remote request” of the server program. If \( e_2 \) happens because of \( e_1 \)’s happening, then \( e_1 \) is a \textit{predecessor (remote predecessor)} of \( e_2 \), and \( e_2 \) a \textit{successor (remote successor)} of \( e_1 \). If \( e_3 \) is the event “receiving acknowledgement” of the same client and happens immediately after \( e_1 \), then \( e_1 \) is a \textit{predecessor of \( e_3 \)}, and \( e_2 \) is a \textit{remote predecessor of \( e_3 \)}. Also, \( e_3 \) is a \textit{successor of \( e_1 \)} and \( e_2 \) a \textit{remote successor of \( e_2 \)}. It is easy to know that \( (E, \preceq) \) is a partially ordered set.

### 3.2. The structure

The tool consists of a controller and a group of managing servers. The controller has two main parts: a user interface (including command interpreter and I/O) and a filter, while a managing server (MS) consists of a server and an event database. Each host which has program parts being
monitored on it has a managing server. The controller can be invoked at any host. By communicating with each related MS, the controller can present the monitored results to the user. Of course, several controllers can be invoked by several users simultaneously. Fig. 5 illustrates the structure of the tool.

There are three steps to use the tool. At the monitoring step, all events that occurred in one host are monitored by the local MS and recorded into the local event database. All events, including communication-oriented calls, process forks and exits in both client and server program parts are monitored. At the ordering step, the partial order among events is established. At this step, each MS exchanges remote predecessor/successor information through the controller and has all remote relationship ordered. Then, the local predecessor/successor relationship is established by each MS over its local event database respectively. The last step is presentation. By combining the results on all related event databases, the filter can present the execution trace of the distributed program at several levels.

4. Implementation issues

4.1. Monitoring library

We have provided a monitoring library which has to be linked with the program being monitored. This library provides replacements for the following BSD4.3 UNIX operating system calls:

- `fork()`: create a child process
- `signal()`: signal a process
- `kill()`: signal a process
- `socket()`: create a socket
- `bind()`: bind a name to a socket
- `listen()`: listen for a connection
- `accept()`: accept a connection request
- `connect()`: initiate a socket
- `closel()`: close a socket
- `write()`: send out a message
- `read()`: receive a message
- `send()`: send out a message
- `recv()`: receive a message
- `sendto()`: send out a message
- `recvfrom()`: receive a message
- `sendmsg()`: send out a message
- `recvmsg()`: receive a message

These replacements first perform some work required by the monitor, such as reporting to the local MS of the occurrence of the event, and then do the normal work of the original call. A preprocessor is used to change all these system calls of a distributed program into corresponding replacement calls. The program is then compiled and linked with the monitoring library. After that, all events will be captured by the tool during the program's execution.

4.2. Managing server

On each host, there is a managing server (MS) which consists of a server and an event database. Each entry of the event database stores an event and includes the following fields:

- `name`: event name. A UID
- `pid`: process ID
- `pgm_name`: program part name
- `m_name`: predecessor event name. A UID array
- `s_name`: successor event name. A UID array
- `fact_info`: fact information
- `message`: message part of the event

The fact information of an event consists of two parts. The first part is a character string that provides readable information on the fact. If the user does not provide fact information, such information will be assigned by the monitoring library; these may be "begin send() call" or "fork new process", etc. Otherwise, the user provided character string is used. The second part contains the position (within the program part) of the statement that carries out the event. This part of the fact information is to be used at the fourth level for locating the source code of an event.

An MS has the following functions:

1. Database management. Responsible for the management of the local event database. The database is protected by the MS and any access of it must go through the MS.

2. Event logging. When an event occurs, it is logged by the local MS into the event database. Because events may occur very fast (or even concurrently), while the logging of an event requires some amount of time, we use an event queue to queue up all events waiting to be inserted into the database. All events are queued into an event queue after they occur, and the local MS looks at the queue and puts the events into the event database.
3. Communicating with the controller. All communications among MSs are conducted by the controller. Many commands can be issued by the controller and performed by MSs. This is the only way a user can access the event database.

Storing all the events of a distributed program execution may cost too much memory, especially for large programs, so facilities are provided to allow the user to select and store events that may be interesting. Also the user can let the tool store only the fact parts of some events and whole information on others.

4.3. Ordering events

As is known, the time system of each host in an LAN is not synchronised, so we can not use the timestamp in Definition 1 to order all events. But the timestamp can certainly be used to order events that occur in one process because they always remain within the same host. That is, events within each process of a distributed program can be fully ordered. But it is impossible to fully order events of different processes. As mentioned earlier, there are some partial ordering relationships among these communication and process fork events. The following steps are used to establish the partial ordering among all events:

1. Communication related predecessors. When a communication related event occurs (for example, the issuing of a sendto() or recvfrom() call), it will cause the occurrence of an event which belongs to another process (and also possibly on another host). In that case, the first event is changed (by the debugging library) to carry not only the original information, but also the event's name. On the other hand, the second event is also changed (by the debugging library) to not only receive the original information, but also the first event's name, and this name is stored by the local MS as the predecessor of the second event.

2. Process fork related predecessors. When a process fork event occurs, it will cause a new process to be set up and executed. This event is changed (by the debugging library) to carry the name of the event, and the first event of the new process will use the carried name as its predecessor event.

3. Form all remote successors. Steps (1) and (2) will establish all remote predecessors. The remote successors are built by each involved MS and the controller. Each MS checks all events in its local database. If its event e has a remote predecessor named f, then the MS will be responsible for storing e as f's successor. This is easy if e and f are in the same database (for example, the fork events); otherwise f's successor will be stored by the communication of these two MSs through the filter.

4. Form all other successors and predecessors. All the events within a process are ordered by their timestamps, and their predecessors/successors are stored using this order. In one process, the predecessor of event e is event d if d.t is immediately less than e.t., and the successor of e is event f if f.t is immediately greater than e.t. This ordering is performed by each MS concurrently.

Using the above method, all communication events can be partially ordered by predecessor/successor relation. For the events within a single process, we can fully order them by their timestamps. Combining these two relations together, we can have some partial ordering over all events of a distributed program.

4.4. Layered presentation

After the monitoring and ordering steps, the user goes to the third step, presentation. A suitable way of understanding a distributed program is to use the top-down method where the structures of the program parts are presented first, then the structure of processes, then events, and finally the contents of the programming code.

At the top level (program level), communication between program parts is displayed. A distributed program may have many program parts. The tool provides a facility that allows the user to select program parts for display. Because in each event entry, there is a field prog_name which specifies the name of the program part (Section 4.2), we use this information and the event ordering to draw the communication diagram between program parts. At this level, no event detail is displayed. The user can have a nice top-view of the program communication.

From the top level, the user may have some idea of which part of the program needs further exploration. So some (or all) relevant processes can be selected and the second level (process level) will display the communication between these selected processes. We also use a field (pid, see Section 4.2) in every event entry and the event ordering to draw the diagram.

Usually from the second level, some processes can be selected for further investigation. The third level (event level) displays the events and communications between selected processes (of course, it can also display all the events relations of the program). At this level, the event numbers are used and the user can consult these numbers with the event tables.

From the third level, the user may identify the events that are essential for analysing and understanding the distributed program. The fourth level will then use a text editor to display the relevant program segments. The user can study the program segments to analyse and understand the details of the communication. We use the second part of information in the fact_info field (see Section 4.2 for details) to locate automatically the position of the program segment.

During the displays, the user can view event details at two levels. At the table level, the events of a selected process are displayed in tabular form (event table). Then the user can select an event within an event table and ask the tool to display its details (detail level, which displays the message part of the event).

190
4.5. Evaluation

An evaluation of the tool was conducted in late 1993. Two groups each of three third-year students were given the source code and executables of a distributed calendar system [22] and were asked to produce a report on the structure of the system within a week. The system has two server programs, two client programs, and some library routines. It has about 3000 lines of code. The first group was given our tool while the second was not. After two days, the first group presented their report, whereas the second group was only able to present their report after five days. The main problem faced by the second group was that they did not know where to start. In contrast, the first group used our tool to analyze the distributed program systematically at four levels.

5. Extensions

The tool described so far can only record events related to system calls listed in Section 4.1. Sometimes a user may be interested in the combination of several events. For example, if a server is going to receive messages from two different processes, it is interesting to see if these two messages are all received during the execution, or to know the order of their arrival. We have extended our tool to deal with this situation. We name the events related to system calls of Section 4.1 as preliminary events and give the following definitions.

Definition 4. Let \( e_1, e_2 : E \) (\( E \) is defined in Definition 2 of Section 3.1) and \( e_1 = (f_1, m_1) \), \( e_2 = (f_2, m_2) \). By \( f_1 * f_2 \) we mean that the happening of \( e_2 \) follows the happening of \( e_1 \), that is, \( e_1 \preceq e_2 \) holds. By \( f_1 \cap f_2 \) we mean that we cannot tell which of \( e_1 \) and \( e_2 \) happened first, that is, there is no predecessor relation exists between these two events. We can view "*" as sequential and "\( \cap \)" as concurrent. By \( f_1 \cup f_2 \) we mean that both \( e_1 \) and \( e_2 \) occur. By \( f_1 \cup f_2 \) we mean that either or both \( e_1 \) and \( e_2 \) occur (if \( e_1 \) does not occur, we denote that as \( \neg e_1 \)). So we have

\[
\begin{align*}
  f_1 * f_2 & \quad \text{if } e_1 \preceq e_2 \\
  f_1 \cap f_2 & = \quad \text{if } e_2 \preceq e_1 \\
  f_1 + f_2 & \quad \text{otherwise}
\end{align*}
\]

and \( f_1 \cup f_2 = f_1 \cap f_2 \) or \( f_1 \) or \( f_2 \). Similarly, by \( m_1 * m_2 \) we mean that message \( m_1 \) is followed by \( m_2 \) and by \( m_1 + m_2 \) we mean that two messages are independent of each other.

Definition 5. If \( e_1 = (f_1, m_1) \) and \( e_2 = (f_2, m_2) \) are events, then

\[
\begin{align*}
  e_1 \cap e_2 & = (f_1 \cap f_2, m_1 \cap m_2), \quad e_1 \cup e_2 = (f_1 \cup f_2, m_1 \cup m_2), \\
  e_1 * e_2 & = (f_1 * f_2, m_1 * m_2), \quad e_1 + e_2 = (f_1 + f_2, m_1 + m_2)
\end{align*}
\]

are also events, where

\[
\begin{align*}
  m_a & = \quad \text{if } e_1 \cap e_2 \\
  m_a & = \quad \text{if } e_1 \cap e_2 \\
  m_a & = \quad \text{otherwise}
\end{align*}
\]

\[
\begin{align*}
  m_b & = \quad \text{if } e_1 \cap e_2 \\
  m_b & = \quad \text{if } e_1 \cap e_2 \\
  m_b & = \quad \text{otherwise}
\end{align*}
\]

We call these new events combined events. Combined events are usually defined by programmers. To be able to record these combined events, we need to expand the event name definition described in Definition 1 of Section 3.1 to include an optional part, called affix. So, the definition of an event name is now \( \langle \text{term}, \text{affix} \rangle \), where the affix is an optional affix name (a character string) defined by the user.

The priority of the above operators is, from high to low, \( \cup, \cap, *, + \). So the expression \( e_1 * e_2 \cap e_3 \cup e_4 + e_5 \) is actually \( (e_1 * (e_2 \cap e_3) \cup e_4) + e_5 \). It can be proved that if \( E \) is the set of all (preliminary and combined) events of a distributed program, then \( (E, \preceq) \) is still a partial order set.

A user uses an event definition file (EDF) to define a combined event. The following is a very simple EDF which defines a combined event BothCreated as the intersection of two preliminary events CreateSock1 and CreateSock2. That is, if both preliminary events happened, then so did the combined event.

BEGIN

preliminary Event:

CreatedSock1, CreatedSock2;

Combined Event:

(BothCreated, "Both sockets are created");

BothCreated = CreatedSock1 \cap CreatedSock2;

END

Several steps are needed to use an EDF. First, the user inserts into the program parts being analysed an affix definition function before each preliminary event which is to be used in the EDF. The format of the affix definition function (defined in the debugging library) is affix\_define(affix), where affix can be any character string. This string will be appended to the preliminary event name when the event happens. Second, the combined events are built using these affix names, and the EDF is read and evaluated by the controller. When any of these affixed preliminary events happens, they are sent to the controller by the local MSSs (of course, the local MSSs also record them as usual). The controller then evaluates the combined events expressions and records them in the event database if any of them is true. The predecessors of a combined event are all the events.
(preliminary and/or combined events) in the right-hand side of the event expression.

6. Summary

This paper describes the design and implementation of a tool for assisting in the analysis and understanding of distributed programs. The tool has several managing servers which record the events of program parts of their hosts into their local event databases. By using an ordering scheme, all events of a distributed program can be partially ordered, and the event graphs in different levels and the relevant event tables can be built. These event graphs and tables are then used to present the program to the user in a top-down manner.

The paper first presents the model of a layered view of a distributed program. This top-down and layered view offers a systematic way of analysing and understanding distributed programs. It then describes the design and implementation of a tool to realise the layered model and to assist in analysing and understanding distributed programs.

References

Using Sensors to Detect Failures in a Loosely Integrated Heterogeneous Database System

Wanlei Zhou
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract In this paper we apply the reactive system concepts in the design and implementation of a fault detection facility for a loosely integrated heterogeneous database system. The database system consists of a number of fully autonomous databases that, through their willingness, can share some portion of their information. Sensors (functions that expose some part of the state of an object) are placed on individual databases and system objects (such as traders, agents, and other sensors). Other databases and system objects, and applications can subscribe to a sensor and therefore are notified whenever the sensor value changes. The fault detection mechanism uses sensors to monitor individual databases and system objects and to detect database or system faults.

Keywords: Distributed database systems, reactive systems, fault-tolerant computing.

1 Introduction

A sensor is a system object of a reactive system that monitors an application's state and reports some aspects of the state to its subscribers. Reactive systems, as defined by Harel and Pnueli [5], are those systems that are repeatedly prompted by the outside world and their role is to continuously respond to external inputs. These systems use sensors and actuators to maintain an ongoing relationship with their environment.

The reactive system concepts are an attractive paradigm for system design, development, and maintenance because it separates policy from mechanism. Several systems, such as Meta [10], DisCo [9] [8] and STATEMENT [4], and languages, such as Reactive C [2] and Reactive Pascal [7] that based on the reactive system concepts have been developed recently.

However, most of the research on reactive systems are concentrated on process control (such as controlling a robot). This paper applies the reactive system concepts in developing distributed fault-tolerant applications.

The development of fault-tolerant software is a very difficult task. One reason is that the fault-tolerant computing policies and mechanisms are deeply embedded into most application programs. Therefore these application programs cannot cope with changes in environments, policies, and mechanisms very well.

By using the reactive system model, we can separated fault detection and fault tolerance processing from applications. This paper shows the flexibility and the modularity of the reactive system model. That is, in an application program, various fault-tolerant policies can be applied and various mechanisms can be implemented.

In this paper we apply the reactive system concepts in the design and a prototype implementation of a fault detection and fault tolerance facility for a loosely integrated heterogeneous database system. The database system consists of a number of fully autonomous databases that, through their willingness, can share some portion of their information. Sen-
sensors (functions that expose some part of the state of an object) are placed on individual databases and system objects (such as traders, agents, and other sensors). Other databases and system objects, and applications can subscribe to a sensor and therefore are notified whenever the sensor value changes. The fault detection mechanism uses sensors to monitor individual databases and system objects and to detect database or system faults. A fault tolerance mechanism can use actuators (functions that can change the state of an object) to react on database or system object faults.

The paper is organised as follows. Section 2 briefly introduces the architecture of the loosely integrated heterogeneous database system. Section 3 presents the fault-tolerant models used in design servers in such a system. Section 4 introduces our reactive system model. Section 5 presents the design and implementation issues. Section 6 describes the fault detection and fault tolerance, and finally, Section 7 summaries the paper.

2 The Loosely Integrated Heterogeneous Database System

Most of the existing approaches in integrating heterogeneous database systems use passive information sharing. That is, they let some authority (e.g., the global schema) decide what are the content and format of information sharing. In [14], we have implemented a loosely integrated heterogeneous database system by using active information sharing. That is, existing database systems are allowed to maintain their autonomy, yet through their willingness, provide a substantial degree of information sharing. The participating database systems themselves decide the content and format of information sharing.

We use a trader to manage the shared information among participating databases. A trader is a third-party object that links clients and servers in a distributed system [6]. By using a trader, servers can advertise (export) their service offers and clients can get (import) information about one or more exported service offers that match some objectives [3] [1]. Traders have been a subject of international standardization for some time [6].

If a database system is willing to share part of its information with other database systems, it exports that information as a service offer to the trader. Application programs (clients) wishing to make use of the shared information have to import such service offers from the trader and then access the database(s) concerned.

There are two types of trading:

- **Direct trading.** The database systems export their service offers directly to the trader. The clients import these service offers from the trader and access the relevant database systems directly.

- **Indirect trading.** The database systems appoint a trading agent to manage the service offers. The trading agent is then responsible for exporting the service offer to the trader. After importing the service offer, the clients call the trading agent and all accesses to the database systems have to go through the trading agent.

Figure 1 depicts the two types of trading processes.

Figure 1: Trading types: direct and indirect trading
A set of operations have been defined for exporters, importers, and system maintenance [14].

Figure 2 depicts the architecture of our loosely integrated heterogeneous database system. Currently two database systems (Oracle and MiniSQL) are running on two separate Sun workstations. A third database server, the Ingres database system, is to be added into the system soon. The trader, trading agents, and user application programs can be executed on any Sun workstations.

Figure 2: Architecture of the loosely integrated system

3 Fault-Tolerant Models for Servers

In [12] and [13] we have proposed four types of fault-tolerant servers, as described below.

A simple server (Figure 3(a)) is an ordinary server possessing of no fault-tolerant features. When a simple server fails, all operations to it have to be aborted.

A service providing server (Figure 3(b)) has a buddy server running (or in a cold stand-by state, i.e., ready to be executed) somewhere in the network, but no communication between the server and its buddy. When a service providing server fails, an operation to this server will be automatically re-directed to its buddy server by the system (the buddy server will be activated if it is in the cold stand-by state).

Figure 3: Server types

An object managing server (Figure 3(c)) also has a buddy running in the network. It manages a critical object that might be updated and shared among clients. An operation to such a server, if it will change the object state, is actually a nested operation.

All buddy servers are simple servers. That means, when a server (service providing or object managing) fails, its buddy server provides alternative service in a simple server manner. Also, when a buddy server fails, a service providing server or an object managing server will be reduced into a simple server.

A twin-server has two servers running in parallel to provide the fault-tolerant service. We call them as twins and denote them as \((T^1, T^2)\). These two servers usually (but not necessarily) locate on different hosts for increasing reliability.

When a client requests a service, the request is mapped to one of the twins. Without loss of generality, suppose that twin server \(T^1\) is selected. The request is then sent to \(T^1\) for processing. At the same time, a copy of the request is also sent to \(T^2\) and is kept in its reserve. If \(T^1\) has completed the service of the request, it notifies \(T^2\) and \(T^2\) will drop the copy of the request from its reserve.

At any time, if one of the twins, say \(T^1\), dies for some reason, the survivor twin \(T^2\) will continue the service of \(T^1\) by using the requests.
kept in its reserve. Then $T^2$ will invoke another server to be its new twin (also named as $T^1$). After that, the normal operation of the twins is resumed. Figure 4 depicts the twin-server model.

![Diagram of twin-server model]

Figure 4: The twin-server model

In our heterogeneous database system, the priorities of fault-tolerance considerations for different servers can be listed, from high to low, as follows: trader, database server, trading manager, and trading agent. Different fault-tolerant policies will be applied to these servers.

- **Trader.** The trader is a key component of the system. The system cannot function once the trader fails. Therefore it requires full fault tolerance.

- **Database servers.** Each individual database server maintains database operations directed to it. If it fails, these operations cannot proceed, but database operations to other database servers are not affected.

- **Trading managers.** Each trading manager maintains trading operations directed to a particular database server. If it fails, these operations cannot proceed, but trading operations to other trading managers and database operations to the database server managed by the failed trading manager are not affected.

4 The Reactive System Model

Figure 5 depicts the architecture of our reactive system model. In this model, sensors are attached to applications to obtain their states (or equivalently, to monitor some events about the applications). These events are sent to the decision making manager (DMM). The DMM reacts to these events by using the actuators to change the states of the applications.

![Diagram of reactive system architecture]

Figure 5: The reactive system architecture

In this model, a sensor reports some aspects of an application's state. A system entity subscribes to a sensor to receive its report. An actuator provides a way to control the behaviour of an application. A decision making manager obtains information about applications via sensors, processes the information, and modifies the states of applications via actuators.

Sensors and actuators can be simple or composite. A simple sensor (simple actuator) can only be directly attached to one (local) application. Figure 6(a) shows a simple sensor (actua-
tor). A composite sensor (composite actuator) can be composed of multiple sensors (actuators) from multiple applications (Figure 6(b)).

![Diagram](https://via.placeholder.com/150)

(a) A simple sensor/actuator  
(b) A composite sensor/actuator

Figure 6: Sensors and actuators

5 Design and Implementation Issues

The first responsibility of sensors is to monitor events. In order to monitor events we have to assign each event a unique name. The following definitions are a simplified version based on [11].

Definition 1. An event name is a text object with the following syntax:

```
eventname ::= EventID.HostID
```

The terminals have the following meaning:

- *EventID* is a unique number (within a local computer).
- *HostID* is an identifier for the host on which the event occurred.

If *en* is an event name, we use *en.i* to denote the event ID and *en.h* to denote the host ID, respectively. Clearly, the combination of *en.i* and *en.h* guarantees the uniqueness of event names throughout the heterogeneous distributed database system.

Definition 2. A simple event is defined as a pair (*en*, *m*) where *en* is an event name and *m* a message. In this context a message is some information associated with the event. For example, we can associate the name of the event that causes this particular event, a time-stamp, or a programmer-defined string used to characterise one or more event attributes with the message part of an event.

Definition 3. Let *e*, *e₁*, and *e₂* be events. The following operations are defined over events:

- **Happens.** We use *e* to mean that event *e* happens within a preset time period.
- **Not happen.** We use ¬*e* to mean that event *e* does not happen within a preset time period. ¬*e* is still regarded as a simple event.
- **Or.** We use *e₁* ∨ *e₂* to represent the fact that either *e₁* or *e₂* or both of them happen.
- **And.** We use *e₁* ∧ *e₂* to represent the fact that both *e₁* and *e₂* happen.

Composite events are made from simple and/or composite events by applying event operations.

In our design, a simple sensor is responsible for monitoring one simple event. According to the way that a sensor performs its work, we have defined the following three types of simple sensors:

- **An event sensor** reports to its subscribers when the monitored event happens.
A timer sensor periodically sends out some message to its subscribers.

A polling sensor periodically checks some state of the application and reports to subscribers.

Simple sensors can be combined together to form composite sensors.

A local event manager (LEM) runs on each host and manages communication activities (such as event reporting) for subscribers and sensors located on remote machines. If an event happens, the event is reported to local and remote subscribers through local and remote methods (Figure 7).

A: Report to local subscribers using UNIX stream sockets
B: Report to remote subscribers using Internet stream sockets

Figure 7: Local event manager

In Figure 7, when the event monitored by sensor S1 happens, the event is reported directly to all the local subscribers, including the local event manager. The LEM is then responsible for propagating the event to all the remote subscribers. The main reason for separating the processing of local and remote event reporting is to make sure that the local subscribers get the sensor's event reporting as fast as possible.

In our design, the DMM is where the application policies are specified and maintained. It uses actuators (procedures that change the state of applications) to perform the required work on applications. For example, an actuator can perform the task of simply setting a value, or the task of activating a buddy server.

The second responsibility of sensors is to report state changes to the subscribers. It is obvious that some form of one-to-many communication facility is required in implementing the reporting function. We use UNIX stream sockets to build local one-to-many communication primitives and the Internet stream sockets to build remote one-to-many communication primitives.

The following one-to-many communication primitives are used in our implementation:

- groupCreateLocal(). It creates a local one-to-many communication group where one object acts as the sender and others the receivers.

- groupDestroy(). It destroys a communication group.

- groupJoinLocal(). An object uses this primitive to join a local one-to-many communication group as a receiver.

- groupLeave(). An object uses this primitive to leave a one-to-many communication group (the object must be a receiver of the group).

- groupSendLocal(). A sender uses this primitive to send a message to all the local receivers of the group.

- groupSendRemote(). A sender uses this primitive to send a message to all the remote receivers of the group.

A simple sensor uses the following function (called the event report function) to report an event:
sensor_report_event(localSubscribers, event)

where,

- localSubscribers contains the local subscribers to this sensor (it is actually a group ID). Local subscribers can be the local DMM, the local event manager, or a local sensor.

- The event is a simple event (en, m).

- The function uses the communication primitive groupSendLocal() to send an event report to all local subscribers.

- The LEM is then responsible for sending the event report to all remote subscribers using the groupSendRemote() primitive.

Simple sensors must be closely attached to their applications. We use the following two methods to create simple sensors according to the types of sensors.

- Event and timer sensors. These sensors are created by inserting into an application program the event report function. This implies that the source code of the application program must be available and the re-compile of the application program is necessary. Most of the servers (the trader, the trading managers, and the trading agents) can use these sensors.

- Polling sensors. These sensors are created as independent processes that run in parallel with their applications. This implies that polling sensors can be used to monitor applications that we do not have their source code (such as the Oracle database server).

The groupCreateLocal() communication primitive is used to create the initial one-to-many communication group (from the sensor to the initial localSubscribers, i.e., the local DMM and the LEM) of each sensor.

The groupJoinLocal() communication primitive is used to subscribe to a sensor locally and the primitive groupJoinRemote() is used to subscribe to a sensor remotely.

The facility for creating a composite sensor has been planned but has not been implemented yet. The principle of creating a composite sensor is to use composite events.

6 Fault Detection and Fault Tolerance

Different types of sensors are used to detect faults in different servers of the system.

The trader is implemented as a twin-server. That is, two trader servers are executed on different computers to provide the service. Event and timer sensors are placed into various positions of the trader (Figure 8).

![Figure 8: Fault detection: the trader](image)

We use \( T^1 \cdot S_j \) to denote the sensor \( S_j \) attached to the trader server \( T^1 \). Since \( T^1 \) and \( T^2 \) are symmetric, we only describe the functions of sensors for \( T^1 \).

The timer sensor \( T^1 \cdot S_1 \) periodically reports to subscribers the liveness of trader server \( T^1 \). So,

\[
\text{if } -T^1 \cdot S_1, \text{ then } T^1 \text{ is faulty.} \tag{1}
\]
The time interval for evaluating this is set to be greater than the time interval of $T^1.S_1$.

The event sensors $T^1.S_2, T^1.S_3, T^2.S_2,$ and $T^2.S_3$ monitor the inter-communication between the twins. $T^1.S_2$ monitors the event where a request for a nested operation from $T^1$ to $T^2$ takes place. $T^2.S_3$ monitors the event where a returning of a successfully executed nested operation by $T^2$ on behalf of $T^1$ takes place. Therefore,

$$\text{if } T^1.S_2 \land \neg T^2.S_3 \text{ then}$$
$$\text{the nested operation fails. (2)}$$

The combination of event sensors $T^1.S_4$ and $T^2.S_5$ provides the possibility of consistency checks for the normal queue of $T^1$ and the reserve queue of $T^2$. $T^1.S_4$ monitors the event where an input to $T^1$'s normal queue takes place, while $T^2.S_5$ monitors the event where an input to $T^2$'s reserve queue takes place. Therefore,

$$\text{if } T^1.S_4 \land \neg T^2.S_5 \text{ then}$$
$$\text{the queues are not consistent. (3)}$$

A polling sensor is attached to each database server to periodically check the liveness of the database server (Figure 9(a)). So,

$$\text{if } \neg S, \text{ then the DB server is faulty. (4)}$$

The time interval for evaluating this is set to be greater than the polling time interval of $S$.

A timer sensor is inserted into each trading manager and each trading agent to report periodically to subscribers (Figure 9(b), (c)). So,

$$\text{if } \neg S, \text{ then}$$
$$\text{the trading manager/agent is faulty. (5)}$$

The time interval for evaluating this is set to be greater than the time interval of $S$.

Various fault tolerance policies are implemented by DMMs through the help of actuators.

On the host that the trader server $T^1$ is running, there is a DMM ($DDM_{T1}$) that subscribes to $T^1.S_1, T^1.S_2,$ and $T^1.S_4$ locally and to $T^2.S_1, T^2.S_2,$ and $T^2.S_3$ remotely (since the two twin-servers running on different hosts). Similarly, on the host that $T^2$ is running, a DMM ($DDM_{T2}$) subscribes to $T^2.S_1, T^2.S_2,$ and $T^2.S_4$ locally and to $T^1.S_1, T^1.S_3,$ and $T^1.S_5$ remotely.

Both $DDM_{T1}$ and $DDM_{T2}$ use formula (1) to monitor the liveness of $T^1$. However, $DDM_{T2}$ takes the initiative to make the decision on what to do after knowing that formula (1) is true. In that case, $DDM_{T2}$ asks $DDM_{T1}$ to confirm that its evaluation of (1) is also true. If so, it asks $DDM_{T1}$ to restart the twin server on the original computer. If $DDM_{T1}$'s evaluation of formula (1) is false, then a second evaluation by $DDM_{T2}$ is carried out. If the second time the two DMMs still disagree with their evaluation, then the twin-server is restarted on another computer. $DDM_{T2}$ also moves all jobs in $T^2$'s reserve queue to $T^2$'s normal queue.

If $DDM_{T1}$ has evaluated that formula (1) is true, then it sets a timer to wait for the initiative action from $DDM_{T2}$ and responses accordingly. If $DDM_{T2}$ does not contact $DDM_{T1}$ within the time limit, $DDM_{T1}$ will start the failed server on the original (local) computer.

Checking the failures of nested operations between $T^1$ and $T^2$ are also the responsibility of the two DMMs. For example, $DDM_{T1}$ subscribes to $T^1.S_2$ and $T^2.S_3$ in order to monitor the nested operations from $T^1$ to $T^2$. It uses formula (2) to decide if a nested operation from $T^1$ to $T^2$ fails. That is, if $T^1.S_2 \land \neg T^2.S_3$ is true,
then a nested operation has been sent out from $T^1$ to $T^2$ but a successfully response has not been issued. In that case, $DMM_{T^1}$ will use an actuator to re-start the nested operation.

The two DMMs are also responsible for checking the consistency of the queues on the two twin-servers. For example, $DMM_{T^1}$ subscribes to $T^1.S_4$ and $T^2.S_5$ to monitor the new requests going into the normal queue of $T^1$ and the reserve queue of $T^2$, respectively. If a request enters the normal queue of $T^1$ but not the reserve queue of $T^2$ (that is, formula (3) is true), then $DMM_{T^1}$ uses an actuator to re-send the request to $T^2$'s reserve queue.

A separate DMM runs on each machine that has a database server, or a trading manager, or a trading agent running on it. The responsibility of the DMM is to monitor the liveness of these servers running on the local machine, by subscribing to the sensors monitoring these servers. If such a server fails (by evaluating formulas (4) and (5)), the DMM will then use an actuator to restart the server.

The policies for dealing with a database server failure and for dealing with a trading manager/agent failure is a little different. Currently, two database servers are running on two different machines. Let us denote the machine that runs Oracle as $M_o$ and the machine that runs MinSQL as $M_m$. We also have a cold stand-by Oracle server on $M_m$ and a cold stand-by MinSQL server on $M_o$. The DMM on $M_o$ monitors the liveness of the MinSQL database server using a similar algorithm as described above. The same algorithm is also used by the DMM on $M_m$ to monitor the liveness of the Oracle database server. The purpose of this is to make sure that the database servers can be restarted even after a host failure.

When a DMM detects a failure of a local trading manager/agent, it then uses an actuator to restart the failed trading manager/agent.

7 Summary

In this paper we have shown an example of using sensors of the reactive system model in detecting failures in a loosely integrated database system. By using sensors and the reactive system model, we have successfully separated fault detection and fault tolerance processing from the applications. The example also shows the flexibility and the modularity of the reactive system model. That is, various fault-tolerant policies can be easily applied and various mechanisms can be easily implemented.

References


plex Environments. Darmstadt, Germany, June 1996.


Detecting and tolerating failures in a loosely integrated heterogeneous database system

W. Zhou

School of Computing and Mathematics, Deakin University, Geelong, Victoria 3217, Australia

Abstract

This article presents the design and prototype implementation of a fault detection and fault tolerance facility based on the reactive system model for a loosely integrated heterogeneous database system running on a network of workstations. The database system consists of a number of fully autonomous databases that, through their willingness, can share some portion of their information. Sensors (functions that expose some part of the state of an object) are placed on individual databases and system objects (such as traders, agents, and other sensors). Other databases and system objects, and applications can subscribe to a sensor and therefore are notified whenever the sensor value changes. The fault detection mechanism uses sensors to monitor individual databases and system objects and to detect database or system faults. The fault tolerance mechanism uses actuators (functions that can change the state of an object) to react on database or system faults. © 1999 Elsevier Science B.V. All rights reserved.

Keywords: Fault-tolerant computing; Reactive systems; Distributed application; Distributed databases systems; Networks of workstations

1. Introduction

A distributed database system running in a network of workstations consists of many hardware and software components that are likely to fail eventually. In many cases, such failures may have disastrous results. It is therefore essential to build distributed database systems that can tolerate component failures.

However, the development of fault-tolerant software is a very difficult task. One reason is that, in normal practice, fault-tolerant computing policies and mechanisms are deeply embedded into most application programs. Therefore these application programs cannot cope with changes in environments, policies, and mechanisms very well. To build fault-tolerant computing systems that can cope with the constant changes in environment and user requirements, it is essential to separate the fault-tolerant computing policies and mechanisms in application programs.

The reactive system concepts are an attractive paradigm for system design, development, and maintenance because it separates policy from mechanism. Several systems, such as Meta [1], DisCo [2] and STATEMENT [3], and languages, such as Reactive C [4] and Reactive Pascal [5] that are based on the reactive system concepts have been developed recently.

However, most of the research on reactive systems is concentrated on process control (such as controlling a robot). This article applies the reactive system concepts in developing a fault-tolerant heterogeneous database system running on a network of workstations.

The article is organised as follows: Section 2 briefly introduces the architecture of the loosely integrated heterogeneous database system and the fault-tolerant models used in design servers in such a system. Section 3 introduces our reactive system model. Section 4 presents the design and implementation issues. Section 5 addresses some performance issues, Section 6 describes the issues on fault detection and fault tolerance, and finally, Section 7 summarises the article.

2. The loosely integrated heterogeneous database system

Most of the existing approaches in integrating heterogeneous database systems use passive information sharing. That is, they let some authority (e.g. the global schema) decide what are the content and format of information sharing. In Ref. [6], we have implemented a loosely integrated heterogeneous database system by using active information sharing. That is, existing database systems are allowed to maintain their autonomy, yet through their willingness, provide a substantial degree of information sharing. The participating database systems themselves decide the content and format of information sharing.
Fig. 1. Trading types: direct and indirect trading.

Fig. 2. Architecture of the loosely integrated system.
Description of the trader

A trader is a third-party object that links clients and servers in a distributed system [7]. By using a trader, servers advertise (export) their service offers and clients can get export information about one or more exported services that match some objectives [8, 9]. Traders have been object of international standardisation for some time [7]. We use a trader to manage the shared information among participating databases. If a database system is willing to re-part of its information with other database systems, it offers that information as a service offer to the trader. Service offers are managed by the trader as trading text. The following three types of service offers are used and can be shared among database systems:

Data. Each database system has a collection of data that might be of interest to other database systems. This information can be exported to the trader as a service offer that can then be accessed by others. This type of service offers is designed for direct data sharing.

Operations. A database system may not wish to share its data directly with other database systems. In this case, an operation can be created and exported to the trader for indirectly sharing this critical data.

Object. An object contains a piece of data and the operations that manipulate the information. It is essentially the combination of the two previous types of service offers.

Any database willing to share its information has to sort relevant service offers for such information to the trader. Application programs (clients) wishing to make of the shared information have to import such services from the trader and then access the database(s) involved.

There are two types of trading:

Direct trading. The database systems export their service offers directly to the trader. The clients import these service offers from the trader and access the relevant database systems directly.

Indirect trading. The database systems appoint a trading agent to manage the service offers. The trading agent is then responsible for exporting the service offer to the trader. After importing the service offer, the clients call the trading agent and all accesses to the database systems have to go through the trading agent.

Fig. 1(a) depicts the direct trading process involving one database system only. In this figure, the database system exports its service offer to the trader; the client exports service offer from the trader and the trader returns the result to the client. The returned offer contains information about the address of the service and the address of the database system that provides the service; the client then accesses the database system directly and the result is returned. Direct trading may not be appropriate in many cases. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then the indirect trading may be used. Fig. 1(b) depicts the indirect trading process that involves two database systems. In this figure, both database systems (DB1 and DB2) are willing to share their information in one service offer. They then appoint an agent and the agent exports the service offer to the trader. The client imports the service offer from the trader and is given the agent's address (and the description of the service, of course). It then calls the agent and obtains the service.

A set of operations have been defined for exporters, importers, and system maintenance. The system has been used in a prototype implementation that provides services on staff/student information stored in databases located in a multi-campus environment [6].

2.2. The implementation architecture

Fig. 2 depicts the architecture of our loosely integrated heterogeneous database system. Currently three database systems (Oracle, Minisql, and Ingres) are running on three separate Sun workstations located in three subnetworks. The trader, trading agents, and user application programs can be executed on any Sun workstations.

A trading manager is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results. By using the trading managers, we can reduce the duplicating part of each trading agent and concentrate on the work of the trading agent such as schema translation and combination.

The trader is implemented as a server that provides export and import operations. The trading agent acts as both a server and a client. To the user program, a trading agent is a server as it provides services that the user program has imported from the trader. However to a trader program, a trading agent is a client because it exports service offers to the trader.

The first version [6] of our distributed database system is implemented through the use of remote procedure calls. The second version [10] of the system is implemented through the use of Java and JDBC. Java's flexibility of coding once and running anywhere nicely suits our needs for a heterogeneous environment. When working on a distributed database system, JDBC frees developers by its goal of being a DBMS-independent and a uniform interface to different data sources. It actually completes part of the work of schema translation which is a key issue dealing with different database systems. When contacting the actual data source, all one has to do is to address the URL (Unified Resource Location) in the network. The lower level data retrieving and manipulating is handled by JDBC drivers. From the user's point of view, which means one single query sentence can be used to access data in all kinds of
sources, such as Oracle, Mini SQL, Ingres, or Access, etc., provided the system being used has installed specific JDBC drivers.

### 2.3. Fault tolerance requirement

In our heterogeneous database system, different servers require different fault-tolerance considerations:

- **Trader.** The trader is a key component of the system. The system cannot function once the trader fails. Therefore it requires full fault tolerance.

- **Database servers.** Each individual database server maintains database operations directed to it. If it fails, these operations cannot proceed, but database operations to other database servers are not affected.

- **Trading managers.** Each trading manager maintains trading operations directed to a particular database server. If it fails, these operations cannot proceed, but trading operations to other trading managers and database operations to the database server managed by the failed trading manager are not affected.

- **Trading agents.** Each trading agent maintains one or a set of trading operations. If it fails, these trading operations cannot proceed, but other (trading and database) operations are not affected.

### 2.4. Fault-tolerant models for servers

Various models have been proposed in building fault-tolerant servers based on server replication [11–13]. In Refs. [14,15] we have proposed four types of fault-tolerant servers, two of them are described below.

A **service-providing server** (Fig. 3(a)) has a buddy server running (or in a cold stand-by state, i.e. ready to be executed) somewhere in the network, but no communication between the server and its buddy. When a service-providing server fails, an operation to this server will be automatically re-directed to its buddy server by the system (the buddy server will be activated if it is in the cold stand-by state).

A **twin-server** (Fig. 3(b)) has two servers running in parallel to provide the fault-tolerant service. We call them as twins and denote them as \( T^1, T^2 \). These two servers usually (but not necessarily) are located on different hosts for increasing reliability.

When a client (e.g. Client 1) requests a service, the request is mapped to one of the twins (e.g. \( T_1 \), the solid line) for processing. At the same time, a copy of the request is also sent to \( T_2 \) and is kept in its reserve (the dashed line). If \( T^1 \) has completed the service of the request, it notifies \( T^2 \) and \( T^2 \) will drop the copy of the request from its reserve.

At any time, if one of the twins, say \( T^1 \), dies for some reason, the alive twin \( T^2 \) will continue the service of \( T^1 \) by using the requests kept in its reserve. Then \( T^2 \) will invoke another server to be its new twin (also named as \( T^1 \)), after which, the normal operation of the twins is resumed.

In our heterogeneous database system, the priorities of fault-tolerance considerations for different servers can be listed, from high to low, as follows: trader, database server,
The major advantage of this model is the separation of the mechanisms for gathering information and the mechanisms for dealing with failures from the policy for making decisions and therefore may lead to a more flexible software architecture that can cope with the changes of environment and user requirements. That is, in an application program, various fault-tolerant policies can be applied and various mechanisms can be implemented. The feasibility and initial advantages of using reactive system concepts in building Java applications has been shown through our earlier work [16].

Sensors and actuators can be simple or composite.

- **A simple sensor (simple actuator)** can only be directly attached to one application. Fig. 5(a) shows a simple sensor (actuator). Here the simple sensor $S$ is attached to the application $A_1$ and reports some state changes of $A_1$ to the DMM. The DMM receives reports from $S$, makes certain decisions according to some predefined policy, and uses the simple actuator $A$ to change the state of the application $A_1$ when it is necessary.

- **A composite sensor (composite actuator)** can consist of multiple sensors (actuators) from multiple applications. For example, Fig. 5(b) shows that the DMM uses a composite sensor $S_c$ to monitor state changes of two applications $A_1$ and $A_2$, and uses a composite actuator $A_c$ to change some state of $A_1$ and $A_2$ when necessary. The composite sensor $S_c$ consists of two simple sensors $S_1$ and $S_2$ that monitor the state changes of application $A_1$, and one simple sensor $S_3$ that monitors the state changes of application $A_2$. Similarly, the composite actuator $A_c$ consists of three simple actuators $A_1$, $A_2$, and $A_3$.

The reasons for having these two types of sensors (actuators) are:

- **Simplified interface between sensors (actuators) and applications.** As each sensor (actuator) can only be associated with one application, the interface between a sensor (actuator) and an application can be easily defined.

- **Efficiency.** Simple sensors can quickly obtain the states of the application because they are directly attached to the application.

- **Modularity.** Complex sensors (actuators) can be built by using the existing simple and/or composite sensors (actuators).

4. Design and implementation of sensors

4.1. Design issues

The main responsibility of sensors is to monitor events. In order to monitor events we have to assign each event a unique name. The following definitions are a simplified version based on Ref. [17].
Definition 1. An event name is a text object with the following syntax:

\[
eventname ::= \text{EventID.HostID}
\]

The terminals have the following meaning:

- \text{EventID} is a unique number (within a local computer).
- \text{HostID} is an identifier for the host on which the event occurred.

If \(en\) is an event name, we use \(en.i\) to denote the event ID and \(en.h\) to denote the host ID, respectively. Clearly, the combination of \(en.i\) and \(en.h\) guarantees the uniqueness of event names throughout the heterogeneous distributed database system.

Definition 2. A simple event is defined as a pair \((en, m)\) where \(en\) is an event name and \(m\) a message. In this context a message is some information associated with the event. For example, we can associate the name of the event that causes this particular event, a time-stamp, or a programmer-defined string used to characterise one or more event attributes with the message part of an event.

Definition 3. Let \(e, \ e_1\) and \(e_2\) be events. The following operations are defined over events:

- \text{Happens}. We use \(e\) to mean that event \(e\) happens.
- \text{Not happen}. We use \(\neg e\) to mean that event \(e\) does not happen within a preset time period. \(\neg e\) is still regarded as a simple event.
- \text{Or}. We use \(e_1 \lor e_2\) to represent the fact that either \(e_1\) or \(e_2\) or both of them happen.
- \text{And}. We use \(e_1 \land e_2\) to represent the fact that both \(e_1\) and \(e_2\) happen.

Composite events are made from simple and/or composite events by applying event operations.

Only the design and implementation of simple sensors are dealt with in this article. In our design, a simple sensor is responsible for monitoring one simple event. Sometimes it is convenient to apply the above event operations to simple sensors. In that case we actually mean that the operations are applied to the simple events that are monitored by these sensors. According to the way that a sensor performs its work, we have defined the following three types of simple sensors:

- \text{Event sensor}. It reports to its subscribers when the monitored event happens.
- \text{Timer sensor}. It periodically sends out some message to its subscribers.
- \text{Polling sensor}. It periodically checks some state of the application and reports to subscribers.

Simple sensors must be closely attached to their applications. We use the following two methods to create simple sensors according to the types of sensors:

- \text{Event and timer sensors}. These sensors are created by inserting into an application program the event report function. This implies that the source code of the application program must be available and the re-compilation of the application program is necessary. Most of the servers in our heterogeneous database system (i.e. the trader, the trading managers, and the trading agents) can use these sensors.

- \text{Polling sensors}. These sensors are created as independent processes that run in parallel with their applications. This implies that polling sensors can be used to monitor applications that we do not have their source code (such as the Oracle and the Ingres database servers).

Using various features of the Java programming language, these sensors have been implemented as both single and multithreaded entities. Also, both multicast datagram and stream-based communication sensors have been implemented, including a hybrid approach using multicast datagrams on each local sub-network, and stream-based communication between sub-networks.

In our design, the DMM is where the application policies are specified and maintained. It uses actuators (procedures that change the state of applications) to perform the required work on applications. For example, an actuator can perform the task of simply setting a value, or the task of activating a buddy server.

4.2. Multicast datagram sensors

Multicast datagram sensors use multicast datagrams to communicate to subscribers. Using a single thread of control, a datagram is sent out onto the sub-network using an address reserved for multicasting, whenever a specific event occurs. Other entities can subscribe to the sensor, simply by creating a multicast socket, and using the join-group and leave-group primitives provided in the Java language.

In order to avoid packets being flooded onto other networks, Java multicast datagrams include a TTL (Time To Live) field in the header to reduce packets unnecessarily circulating and congesting networks. The programmer can increase the TTL allowing the packet to travel further.

However, many routers provide only limited support for multicasting. Packets are allowed to be multicast within a sub-network, but not to pass through the router into other sub-networks. To overcome this, a tunnelling approach has been implemented.

Tunnelling (as shown in Fig. 6) establishes two software entities, one located on the sensor’s sub-network, and another on the remote subscriber’s sub-network. On the sensor’s sub-network, a courier joins the multicast group, and retransmits the locally multicast packets using either datagrams or sockets to another remote sub-network. At
In order to synchronize the transmission of reports, each sensor creates a ThreadGroup, to which each new thread created is placed into. Using the ThreadGroup, the sensor can invoke each thread to report to its subscribers at the same time, rather than the threads independently having to monitor for events, and report to subscribers. In addition, the ThreadGroup approach places the monitoring of events in one place, within the sensor, rather than each thread having to monitor for the event which duplicates processing. Fig. 7 shows the architecture of a stream-based sensor.

Unlike the multicast sensors, stream-based sensors consist of multiple objects interacting with each other. These are a main sensor object that monitors for events, a listener object that accepts connections from subscribers to the sensor and a collection of zero or more member objects, each representing a connection to the sensor and is responsible for communication between the sensor and the subscriber.

5. Performance evaluation for sensors

The following summary outlines a series of tests and results used to evaluate the performance of each type of sensor. The purpose of these tests is to determine the time taken for each sensor to report an event to subscribers located locally on the same host, on the same sub-network, and also remotely on other sub-networks.

Workstations (Sun Sparc workstations running the Solaris operating system) of each sub-network are linked through a 10 Mbit Ethernet network. These sub-networks (about 100 km apart) are linked through a 34 Mbit ATM Backbone (the implication is that normally the ATM Backbone will not be the bottleneck of the communication during our tests). Sun's JDK1.1.2 Java interpreter was used to run both the sensors and the subscribers.
Table 1
Reporting time (single subscriber). LH: local host; LS: local sub-network; RS: remote sub-network

<table>
<thead>
<tr>
<th></th>
<th>Timing LH</th>
<th>LS</th>
<th>RS</th>
<th>Event LH</th>
<th>LS</th>
<th>RS</th>
<th>Polling LH</th>
<th>LS</th>
<th>RS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multicasting</td>
<td>3.334</td>
<td>4.244</td>
<td>12.234</td>
<td>4.327</td>
<td>5.043</td>
<td>11.080</td>
<td>110.950</td>
<td>120.416</td>
<td>131.775</td>
</tr>
<tr>
<td>Stream-based</td>
<td>3.842</td>
<td>5.264</td>
<td>6.102</td>
<td>4.004</td>
<td>5.354</td>
<td>6.536</td>
<td>119.721</td>
<td>121.461</td>
<td>124.878</td>
</tr>
</tbody>
</table>

5.1. Test descriptions

The tests are conducted on two groups of three sensors, the first group using multicasting datagrams to report to subscribers and the second group using stream-based communication. Within each of these groups is a timing sensor, polling sensor and an event sensor.

The role of the timing sensor is to report to subscribers at discrete intervals of time. By setting the time required for sensing activity to zero (i.e. no sensing activity is performed) tests show how long a sensor takes to send a message, without the overhead of the sensing activity.

The polling sensor tested is required to report on the liveliness of a sleep process running in the background on the same host (of course, the sensing activity can be any predefined task). The sensor determines whether the process is running and reports its state to the subscriber. These tests measure the overhead of checking the liveliness of a process and the time taken to report this liveliness to the subscriber.

The event sensor tested is embedded within a test application which triggers the sensor to report to its subscribers when an empty event has occurred. These tests measure the overhead of the sensor being triggered by an application and the time taken to report this event to the subscriber.

Each sensor is tested 1000 times with an average time taken from these results.

Using the above method, the multicasting sensors are tested reporting to a single subscriber. For every event detected, a multicasting sensor sends out one message, which can be received by all subscribers on the same sub-network almost simultaneously. Therefore, increasing the number of subscribers during testing would have had a marginal influence on the results.

However, the stream-based sensors use a separate thread to manage each connection between the subscriber and the sensor. As an event is detected, the sensor invokes each thread to report to its subscriber. In tests with multiple subscribers, each thread measures the time to report to its subscriber, which is added to a total and divided by the number of subscribers to the sensor. This provides an average overhead associated with increasing the number of subscribers to these types of sensors.

Fig. 8. Local host and local sub-network testings (EventL.H.dat: event sensors running on a local host; TimingL.H.dat: timing sensors running on a local host; EventL.S.dat: event sensors running on a local sub-network; TimingL.S.dat: timing sensors running on a local sub-network).
Results and discussions

Table 1 shows the time (in milliseconds) needed by various types of sensors to report to a single subscriber. In this table, we can see that the multicasting sensors show an expected gradual increase in reporting time as the subscriber is located further and further away from the sensor. Once the subscribers are located on a sub-network remote to the sensor's sub-network, an added overhead of tunnelling the datagram from one sub-network to another is recorded.

Multicast datagrams are an unreliable method of reporting events, and therefore may have limited application in distributed systems covering multiple sub-networks. However, within a local sub-network, the multicasting sensors are the fastest to report events and are not affected by an increasing number of subscribers.

Stream-based sensors also show a gradual increase in reporting time as the distance between the sensor and the subscriber increases. However, no significant increase in the reporting time is recorded when the subscriber is moved from a local sub-network to a remote sub-network because there is no tunnelling overhead. Based on this result, the use of stream-based sensors in reporting events, these sensors may be more suited to distributed systems than the multicasting sensors.

Throughout all the tests conducted, the polling sensor is never to report to subscribers by approximately a factor of 1. These results can be attributed to the execution of a sleep call to determine if a process is alive on the local host. This delay may limit such a sensor's application, such as in hard real-time systems.

The following results show the effect of multiple subscribers on the reporting times when stream-based timing and event sensors are used to report events to 1 through 10 subscribers. Fig. 8 shows the results of local host testing (where all timing and event sensors and their subscribers are located on the same host) and local sub-network testing (where all timing and event sensors and their subscribers are located on the same sub-network).

Fig. 9 shows the result of local host testing and remote sub-network testing (where all timing and event sensors and their subscribers are located on two interconnected sub-networks).

The two figures (Figs. 8 and 9) confirm our intuition that as the number of subscribers increases, the time used by sensors to report to their subscribers increases as well. However, such an increase of time is not very dramatic in cases of a local sub-network and remote sub-networks. This is good news for us as most of the DMMs require reports from sensors located in different hosts. Apart from that, we also have the following two observations:

- The first observation from Figs. 8 and 9 is that the stream-based event sensors take slightly more time to report to subscribers than their corresponding timing sensors. This is because that an event sensor has to be triggered within an application, while timing sensors fire their reports automatically.
- The second observation is that in the case of 1 subscriber,
sensors running on a local sub-network or on remote sub-networks use more time to report than sensors running on a local host. When the number of subscribers increases to 2 and then to 3 (3 then 4 in the case of remote sub-networks), there is very little difference in reporting time. However, when the number of subscribers increases further, the sensors running on a local sub-network or on remote sub-networks use less time to report than sensors running on a local host! This may be due to the testing methods we used: we run each subscriber on an individual host for local and remote sub-networks testing, while for local host testing, all sensors and subscribers are executed on the same host.

6. Fault detection and fault tolerance

In this section we place the sensors we have developed in the previous sections into our heterogeneous database system to detect the possible failures of various system objects. We then use decision-making managers to implement various fault tolerance policies.

6.1. Fault detection

Three types of sensors—event sensor, timer sensor and polling sensor—are used in different servers. These sensors can detect various types of failures of the system.

* The trader. The trader is implemented as a twin-server. That is, two trader servers are executed on different computers (located in different sub-networks, as shown in Fig. 2) to provide the service. Event and timer sensors are placed into various positions of the trader (Fig. 10). We use $T^1.S_1$ to denote the sensor $S_1$ attached to the trader server $T^1$. As $T^1$ and $T^2$ are symmetric, we only describe the functions of sensors for $T^1$.

The timer sensor $T^1.S_1$ periodically reports to subscriber the liveliness of trader server $T^1$. Hence,

$$\text{if } \sim T^1.S_1, \text{ then } T^1 \text{ is faulty},$$

(1)

The time interval for evaluating this is set to be greater than the time interval of $T^1.S_1$. The event sensors $T^1.S_1$, $T^1.S_2$, $T^2.S_2$, and $T^2.S_2$ monitor the inter-communication between the twins. $T^1.S_2$ monitors the event where request for a nested operation from $T^1$ to $T^2$ take place. $T^2.S_2$ monitors the event where a returning of a successfully executed nested operation by $T^2$ on behalf of $T_1$ takes place. Therefore,

$$\text{if } T^1.S_2 \sim T^2.T_3, \text{ then the nested operation fails},$$

(2)

The combination of event sensors $T^1.S_2$ and $T^2.S_2$ provides the possibility of consistency checks for the normal queue of $T^1$ and the reserve queue of $T^2$. $T^1.S_2$ monitors the event where an input to $T^1$'s normal queue takes place, while $T^2.S_2$ monitors the event where an input to $T^2$'s reserve queue takes place. Therefore,

$$\text{if } T^1.S_2 \sim T^2.S_2, \text{ then the queues are not consistent},$$

(3)

- The database servers. A polling sensor is attached to each database server to periodically check the liveness of the database server (Fig. 11(a)). Hence,

$$\text{if } \sim S, \text{ then the DB server is faulty},$$

(4)

The time interval for evaluating this is set to be greater than the polling time interval of $S$.
- Trading managers and trading agents. A timer sensor is inserted into each trading manager and each trading
agent to report periodically to subscribers (Fig. 11(b) and (c)). Hence,

if $\neg S$, then the trading manager/agent is faulty. \hfill (5)

The time interval for evaluating this is set to be greater than the

time interval of $S$.

\subsection*{Fault tolerance}

Various fault tolerance policies are implemented by DMMs through
the help of actuators. On the host that the server $T^1$ is running, there is a
DMM (DMM$_{T^1}$) that subscribes to $T^1$.S$_1$, $T^1$.S$_2$, and $T^1$.S$_4$ locally
and to $T^2$.S$_1$, $S_1$ and $T^2$.S$_3$ remotely (as the two twin-servers running
different hosts). Similarly, on the host that $T^2$ is running, DMM (DMM$_{T^2}$)
subscribes to $T^2$.S$_1$, $T^2$.S$_2$, and $T^2$.S$_4$ locally and to $T^1$.S$_1$, $T^1$.S$_3$
and $T^1$.S$_5$ remotely.

Both DMM$_{T^1}$ and DMM$_{T^2}$ use formula (1) to monitor the
effectiveness of $T^1$. However, DMM$_{T^1}$ takes the initiative to
make the decision on what to do after knowing that formula (1)
is true. In that case, DMM$_{T^1}$ asks DMM$_{T^2}$ to confirm that
the evaluation of (1) is also true. If so, it asks DMM$_{T^1}$
to start the twin server on the original computer. If DMM$_{T^1}$'s
evaluation of formula (1) is false, then a second evaluation
of DMM$_{T^1}$ is carried out. If the second time the two DMMs
agree with their evaluation, then the twin-server is
started on another computer. DMM$_{T^2}$ also moves all jobs
$T^2$'s reserve queue to $T^2$'s normal queue.

Listing 1 shows the algorithm liveness_checking_\_DMM2\_to\_T1() used by DMM$_{T^2}$ for checking the
effectiveness of $T^1$. The algorithm used by DMM$_{T^1}$ for checking
the liveness of $T^2$ is symmetric.

Listing 1: Algorithm used by DMM$_{T^2}$ for checking $T^1$

liveness_checking_\_DMM2\_to\_T1() {
  int checkFlag = 0;
  while (true) {
    /* receive report from $T^1$.S$_1$ */
    if evaluated by DMM$_{T^1}$: $\neg T^1$.S$_1$ = true then
      ask DMM$_{T^2}$ to evaluate: $\neg T^1$.S$_1$;
    if evaluated by DMM$_{T^2}$: $\neg T^1$.S$_1$ = true then
      /* now both DMMs agree with each other that $T^1$ fails */
      ask DMM$_{T^2}$ to restart the twin-server locally;
      moves all jobs in $T^2$'s reserve queue to $T^2$'s
      normal queue;
      checkFlag = 0; /* reset the flag */
      continue; /* go back to while */
    endif;
    /* comes here if both DMMs do not agree with
      each other */
    if checkFlag = 0 then /* first time */
      checkFlag = 1; /* set the flag */
      continue; /* go back to while */
    else /* second time, and both DMMs do not
      agree with each other */
      restart the twin-server on a different computer;
      moves all jobs in $T^2$'s reserve queue to $T^2$'s
      normal queue;
      checkFlag = 0; /* reset the flag */
      continue; /* go back to while */
    endif;
  } /* while */
}

If DMM$_{T^1}$ has evaluated that formula (1) is true, then it
sets a timer to wait for the initiative action from DMM$_{T^2}$ and
responses accordingly (as expected in liveness_checking_\_DMM2\_to\_T1()). If DMM$_{T^2}$ does
not contact DMM$_{T^1}$ within the time limit, DMM$_{T^1}$
will start the failed server on the original (local) computer.

Checking the failures of nested operations between $T^1$
and $T^2$ are also the responsibility of the two DMMs. For example,
DMM$_{T^1}$ subscribes to $T^1$.S$_1$ and $T^1$.S$_2$ in order to
monitor the nested operations from $T^1$ to $T^2$. It uses formula
(2) to decide if a nested operation from $T^1$ to $T^2$ fails. That is,
if $T^1$.S$_1 \land \neg T^2$.S$_3$ is true, then a nested operation has been
sent out from $T^1$ to $T^2$ but a successful response has not
been issued. In that case, DMM$_{T^1}$ will use an actuator to re-
start the nested operation.

The two DMMs are also responsible for checking the
consistency of the queues on the two twin-servers. For example,
DMM$_{T^1}$ subscribes to $T^1$.S$_1$ and $T^2$.S$_2$ to monitor
the new requests going into the normal queue of $T^1$ and the
reserve queue of $T^r$, respectively. If a request enters the normal queue of $T^r$ but not the reserve queue of $T^r$ (that is, formula (3) is true), then DMM uses an actuator to re-send the request to $T^r$'s reserve queue.

A separate DMM runs on each machine that has a database server, or a trading manager, or a trading agent running on it. The responsibility of the DMM is to monitor the liveliness of these servers running on the local machine, by subscribing to the sensors monitoring these servers. If such a server fails (by evaluating formulas (4) and (5)), the DMM will then use an actuator to restart the server.

The policies for dealing with a database server failure and for dealing with a trading manager/agent failure is a little different. When a DMM detects a failure of a local trading manager/agent, it simply uses an actuator to restart the failed trading manager/agent. However, the fault-tolerant policy for database servers is more complex. Currently, three database servers are running on three different machines. Let us denote the machines that run current Oracle, MiniSQL and Ingres servers as $M_n$, $M_m$, and $M_s$ respectively. We also have cold-stand-by Oracle, MiniSQL and Ingres servers on machines $M'_n$, $M'_m$ and $M'_s$, respectively. The DMM on $M_n$ monitors the liveliness of the MiniSQL and Ingres database servers using a similar algorithm as in Listing 1. The same algorithm is also used by the DMM on $M_m$ to monitor the liveliness of the Oracle and Ingres database servers, and by the DMM on $M_s$ to monitor the liveliness of the MiniSQL and Ingres database servers. The purpose of this arrangement is to make sure that the database servers can be restarted even after a host failure.

7. Summary

In this article we have developed a reactive system model for building fault-tolerant applications. We have implemented the model in Java language and measured its performance partially. The model is then applied to the fault detection and fault tolerance for a heterogeneous distributed database system.

The first contribution of this article is the development of a three-layer reactive system model for building fault-tolerant applications. The major advantage of the proposed model is the separation of mechanisms for generating information and for dealing with failures from the policies for making decisions. This separation allows us to apply various fault-tolerant policies that use various fault detection mechanisms and therefore may lead to a better software architecture to cope with changes in environment and user requirements.

The second contribution of this article is the implementation, evaluation, and application of our reactive system model. Implementing the model in Java gives us the potential flexibility of "coding once and running everywhere".

The performance evaluation of sensors implemented Java gives us an indication of how much overhead is associated with the use of a Java sensor. The application of our reactive system model in the fault detection and fault tolerance of a heterogeneous distributed database system shows the feasibility and potential benefits of using our reactive system model in building various fault tolerant applications.

References

Chapter 21

Surviving Network Partitioning in a Web-Based Application

Wanlei Zhou, School of Computing and Math., Deakin University, Australia 3168, wanlei@deakin.edu.au

Abstract

Web-based applications are vulnerable to network partitioning failures. In this paper we present a strategy that allows every part of a Web-based application to continue its operations during a network partitioning. When the network partitioning is recovered, a reconciliation process will bring the system to a consistent state.

Keywords: Web-based application, Network partitioning, Replication, Databases.

1 Introduction

One of the challenges of Web-based systems is the withstand of network partitioning failures [Meliar-Smith and Moser 1998]. A network partitioning splits the network into two or more disjoint parts. Processes of a Web-based application within the same part can communicate with each other, but they cannot communicate with processes of the application located in other parts. In order for the application to be continuously operational, data and processes must be replicated in the network. However, application processes may perform some incompatible operations that can result in inconsistent data during the network partitioning. The challenge is to let the application continue its operations during a network partitioning, yet to reconcile the effects of incompatible operations when the communication is restored. In this paper we have developed a strategy that allows every part of a Web-based application to continue its operations during a network partitioning. When the network partitioning is recovered, a reconciliation process will bring the system to a consistent state.

2 The Application

The application is a prototype of a Web-based sales system spread across three cities: Melbourne, Sydney and Geelong. Each city has a database which records the inventory of the regional warehouse and a group of salespersons that rely on the database for their sales activities. Each city also stores replications of databases of other cities. Therefore, a salesperson can also sale things stored in other cities’ inventory databases, although majority of sales will be from the local inventory database. Figure 1 depicts the architecture of the Web-based sales system.

![Architecture of the Web-based sales system](image)
A transaction management system (consisted of a set of transaction managers) runs between the user (a salesperson and the databases. Java embedded browsers are chosen as the interface between user, which is the front end, and our transaction management system which is a set of Java implemented programs. Java applets in a Web page, triggered by a user when viewed by the Web browser, are links to execute certain transaction operations. In the back end, we use Java Database Connectivity (JDBC) for the servers in the transaction management system to access the physical data sources in various locations. Figure 2 shows the information flow in our Web-based design for the sales system.

Salespersons which have access to local Web servers use the Web Browser to locate a Web page designed for our system. There are Java applets embedded in the page containing fields for user input and for displaying feedback. After selecting his/her transaction service, the salesperson sends the request through the Web which actually triggers the execution of server programs on the fly. Transaction managers in Figure 2 receive user requests and execute them on individual databases. These transaction managers are also responsible for the preparation of updates to replicas.

On the transaction managers’ side, we use Java’s Remote Method Invocation (RMI). The transaction managers in our system are remote server objects listening for calls from applets all the time. They have the detailed information of the server groups which actually execute individual sub-transactions included in a user’s request. When a transaction manager receives a call from the applet, it first decides which server groups to contact. It then acts as a secondary client to invoke those server groups using RMI. After the result is returned to the transaction manager, it passes on to the applet to display to the user and do the necessary job of coordinating among replicas. The server groups are the transaction managers for the primary/non-primary replicas of the same database which are stand-alone Java programs sitting on different machines. The JDBC driver objects are embedded in these programs.

3 Replication Strategy: Normal Situations

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a Web-based computing system [Baentsch et al. 1996]. However, although considerable research effort has been directed towards the design of replication-control protocols, replication is still viewed as a “necessary evil” [Triaftafillou and Taylor 1995]. Most existing replication-control protocols are either inefficient or too complicated to be implemented [Ceri et al. 1991, Chen and Pu 1992]. Reference [Heal et al. 1996] gives a comprehensive overview of replication techniques and annotated bibliographies of selected literature on replication techniques and example systems.

3.1 Transactions

We first define some terms. The Sydney inventory database on the Sydney site is called the primary database (or primary, in short) of Sydney site. Similarly, the Melbourne inventory database is called the primary database of Melbourne site and the Geelong inventory database is called the primary database of Geelong site. Other databases (replicas) stored on a site are called replica databases (or replicas, in short). If a transaction issued from a site (i.e., a salesperson of that site originated the transaction on behalf of a customer) only accesses its primary database, then we call such a transaction as a primary transaction. A transaction issued from a site accesses data of a replica database is called a replica transaction. Transactions propagated from
other sites and to be executed on the current site are called propagated transactions (naturally, we may have propagated primary transactions and propagated replica transactions). Without loss of generality, we define all transactions to be update-oriented transactions.

We also assign a site priority to transactions issued from each site. A transaction issued from Melbourne site has the highest site priority (numbered 0) since the Melbourne site is the location of the business headquarter. Transactions issued from Sydney site have medium site priority (numbered 1) and transactions issued from Geelong site have the lowest site priority (numbered 2).

The execution of a transaction on a primary database can have three results. First, the transaction's execution can be completely successful. For example, if a transaction requires the sale of 10 HP1100 LaserJet printers and the primary inventory database has 20 in stock, then the transaction can be successfully committed. Second, the transaction's execution is not successful initially, but can become successful by the generation of a fulfillment transaction [Melliar-Smith and Moser 1998]. For example, if a transaction requires the sale of 20 HP1100 LaserJet printers and the primary inventory database only has 10 in stock, then a fulfillment transaction will be generated to order 10 (or maybe 15 if the lower limit of the stock is 5) more HP1100 LaserJet printers. In this case, the transaction is also regarded to be successful. The execution of a fulfillment transaction will be carried out before the execution of the original transaction and it follows the same strategy of the execution of a normal transaction. We regard the two successful cases as the same. Last, the transaction cannot be executed successfully and a fulfillment transaction cannot be generated. For example, if a transaction requires the sale of 10 HP1100 LaserJet printers, the primary inventory database has none in stock and the manufacturer has stopped producing the product, then the transaction fails.

The execution of a transaction on a replica database can also have three results. First, the execution can be completely successful (commit), which means that the transaction is executed successfully on the replica database and has been successfully executed on the primary database. Second, the transaction has been successfully executed on the replica database, but the execution result on the primary database is still unknown. We call this transaction as a partially committed transaction. Third, the transaction fails on the replica database. A partially committed transaction will be upgraded into commit if the execution on the primary database is successful, or will be down-graded into fail if the execution on the primary database fails.

### 3.2 Normal operations

We discuss the operations of a transaction manager on a site when there is no failure. Users connected to each site can request two types of transactions, the primary transactions and the replica transactions, to the transaction manager. Also, the transaction manager can receive two types of transactions, the propagated forced transactions and the propagated replica transactions, from other transaction managers. In addition to these transactions, the transaction manager can also receive returned results about the execution on propagated transactions.

![Diagram](image)

**Figure 3: Processing of a primary transaction.** PT: primary transaction, TM: transaction manager
Si: Site i. PFT: propagated forced transaction. PDI: primary database.

For each site, upon originated a primary transaction (PT), it processes the PT right away and returns the result back to the user (Figure 3). If the primary transaction has been successfully executed locally, then the transaction manager on the site propagates the primary transaction to other two sites. The transaction managers on the other two sites will force these propagated primary transactions (PFT) to be executed on their replicas.

161

217
Upon originated a replica transaction (RT), the transaction manager on a site processes the replica transaction using its primary database (if the transaction also accesses the primary database) and the replica databases (Figure 4). If the processing is successful, the transaction is regarded as partially committed and the user is notified of the tentative execution result. If the partially committed transaction accesses one replica, the transaction is sent to the primary site of the replica as a propagated replica transaction (PRT) for further processing. A forward mark is attached to this transaction noting that the third replica should execute this transaction as a propagated primary transaction if the primary site's execution is successful. If the partially committed transaction accesses two replicas, it is sent to the primary site with a higher priority. A forward mark is attached to the transaction to note that the primary site with a lower priority should execute the transaction as a propagated replica transaction.

![Diagram of Processing a Replicated Transaction](image)

Upon receiving a propagated forced transaction (PFT), the transaction manager on a site executes the transaction on the relevant replica database(s). This type of transactions have the highest priority and will be executed immediately (i.e., before any replica transactions and any propagated replica transactions).

Upon receiving a propagated replica transaction (PRT), the transaction manager on a site executes the transaction (Figure 5). If the execution of the transaction is successful and the transaction's forward marked is for this site, then an OK result is returned to the sender of this transaction. If the execution is successful and the transaction's forward marked is for the third site, then the transaction will be sent to the third site as a propagated primary transaction or as a propagated replica transaction, depending on the nature of the forward mark. If the execution fails, then all effect of the transaction will be cleared from this site and a failed result will be returned to the sender of this transaction.

![Diagram of Processing a Propagated Replicated Transaction](image)

Upon receiving a returned result on a propagated replica transaction, the transaction manager on a site will analyse the result (Figure 6). If it is an OK and this site originated the transaction, then the transaction (which was partially committed) will finally commit (i.e., the effect of the transaction will be made permanent and the user is notified of the result). If this site did not originate the transaction, then the effect of the transaction will be made permanent on this site and the OK result will be sent to the origin of the transaction (since there are only three sites in our system). If the result is failed and this site originated the transaction, then all effect of the partially committed transaction will be cleared from this site and a failed result will be returned to the user. If this site did not originate the transaction, then all effect of the transaction will be cleared from this
site and a failed result will be sent to the origin of the transaction.

Figure 6: Processing of a returned result: T transaction

The successful execution of a transaction on a primary database may generate a new fulfilment transaction, in that case, the effect of the fulfilment transaction will be enclosed into the original transaction. For a primary transaction and a replica transaction (if it also accesses the primary database), this can be done easily since they have not involved other sites yet. For a propagated primary transaction, there is no such issue. The processing of a propagated replica transaction involves more work. Let us examine an extreme case. For example, if a replica transaction originated from Sydney site has been successfully executed on both Melbourne and Geelong replicas, then it will be sent to the Melbourne site first (since it has a higher priority) with a forward marking noting that it has to be sent to the Geelong site as a propagated replica transaction. Now assume that when the propagated replica transaction is executed on the Melbourne site, it generates a fulfilment transaction. Then this fulfilment transaction will be sent back to Sydney site as a propagated primary transaction. The effect of the fulfilment transaction will also be incorporated into the original transaction before it is forwarded to the Geelong site. If the execution of the transaction on Geelong site also generates a fulfilment transaction, then the fulfilment transaction will be sent to both Melbourne and Sydney sites as a propagated primary transaction. The generation, incorporation, and propagation of fulfilment transactions are an integrated part of our transaction management system.

4 Replication Strategy: Network Partitioning and Recovery

4.1 Operations during network partitioning

When one site is partitioned from other two sites, then it cannot propagate its transactions to other sites, nor it is able to receive any transactions from other sites. If this site originates a primary transaction and the execution of the transaction is successful, then the primary transaction is logged into a stable storage (a hard disk file) together with a local timestamp (all logged transactions require a local timestamp to keep the local ordering). A destination mark is also attached to the transaction noting that it has to be executed on the two replicas. The transaction is considered committed even the system has to propagate the logged primary transaction to the two replicas when the network partitioning is recovered.

If this site originates a replica transaction and the execution is successful, then the replica transaction is logged into the stable storage. The transaction is considered partially committed. The final fate of the transaction can only be decided when the network partitioning is recovered and the logged replica transaction is executed by the relevant primary site(s). The destination mark of the transaction can be decided using the same strategy for processing replica transactions (as described in Section 3.2).

When two sites are partitioned from the third site, then the two sites can communicate with each other but they cannot communicate with the third site. If a site originates a primary transaction and the execution of the transaction is successful, then the primary transaction will be propagated to the communicating site. The destination mark of the transaction is set to the third site and the transaction is logged into the stable storage.

If a site originates a replica transaction and the execution is successful, then the transaction is considered partially committed. If the transaction can be sent to the communicating site according to the strategy described in Section 3.2, then it is sent to the communicating site with a correct forward mark for further
propagation (either as a propagated primary transaction or as a propagated replica transaction). Otherwise the transaction (with a correct forward mark) is logged into the stable storage and the destination mark of the transaction is set to the disconnected site.

If a site receives a propagated primary transaction, the transaction will be executed the same way as if the network partitioning were not present.

If a site receives a propagated replica transaction, the transaction manager on that site will execute the transaction. If the execution is successful, then the transaction will be logged into the stable storage, together with the original forward mark and a destination mark for the disconnected site. If the execution fails, then all effect of the transaction will be cleared from this site and a failed result will be returned to the sender of this transaction.

If a site receives a returned failed result on a propagated replica transaction, it knows that the communicating site failed to execute the transaction. It then clears all effect of the partially committed transaction from this site and a failed result will be returned to the user.

4.2 Operations during recovery

When a network partitioning recovers, all sites will go into the recovering state where a set of recovery operations will be executed before any new transactions can be executed. The recovery process has the following three stages:

- **Propagation stage.** At this stage all sites group all their logged transactions into two sets according to the destination mark of each transaction. Each set will be sent to the relevant site as a single log message. If a site does not have any logged transactions for a particular site, then a log message of an empty set will be sent to the latter site instead.

- **Ordering stage.** When a site has received log messages from other two sites, it first orders all propagated primary transactions according to their site priority. Transactions with the same site priority are ordered according to their local timestamps. Then all propagated replica transactions will be ordered the same way as the propagated primary transactions: i.e., firstly the site priority then the local timestamps.

- **Execution stage.** After the ordering of all transactions, the transaction manager on a site will execute these transactions according to their ordering. Firstly the propagated primary transactions are executed. The execution results of these transactions must be OK since these transactions have been successfully executed on their primary sites. Then the propagated replica transactions are executed according to the ordering.

The strategy for the execution of the propagated replica transactions during the execution stage is very similar to the strategy described in Section 3.2. For example, if the execution of a propagated replica transaction is OK and the forward mark of the transaction is for the current site, then the transaction commits and an OK result is returned to the sender of this transaction. When receives the OK result, the sender of this transaction will commit the transaction. If the sender is also originated the transaction, it will notify the user (using an asynchronous mechanism) that the partially committed transaction has been upgraded to fully commit.

The execution of propagated replica transactions may also generate fulfillment transactions. These fulfillment transactions will be processed using the same strategy as described in Section 3.2.

All databases stored on each site will be in a consistent state when these sites complete the recovery process.

5 Conclusions

In this paper we have presented a Web-based application that can survive network partitioning failures. Our
major contributions are: (a) introduction of a partial commit state for transactions on replicated databases, (b) introduction of a fulfilment transaction as a part of normal database operations, and (c) combination of partially commit states and fulfilment transactions in dealing with network partitioning failures.

The concept of partially committed transactions was introduced in [Zhou and Molinari 1996] in dealing with remote procedure call transactions. The extension of this concept into replicated database transactions allows a database transaction to be proceeded even in a network partitioning. The concept of fulfilment transactions was introduced in the Totem project [Melliar-Smith and Moser 1998] to deal with network partitioning. The extension of this concept as a part of normal database operations not only allows fulfilment transactions to be used to reduce the effect of incompatible operations during a network partitioning, but also captures the essence of business transactions.

References


Toward the Building of Component-Based Electronic Commerce Systems

Wanlei Zhou, Naomi Augar and John Casey
School of Computing and Mathematics, Deakin University, Clayton, Melbourne, VIC 3168, Australia.
wanlei@deakin.edu.au

Abstract
The World Wide Web and the Internet have already changed the way that many companies conduct their business and the demand for electronic commerce software is growing exponentially. Electronic commerce software must be very flexible and interoperable because business practices vary tremendously among companies and the need of exchanging services among companies and customers. To meet these requirements, developers are recognizing that they need a standard, common framework. This paper tries to apply the component-based software development methodologies to develop a framework for developing e-commerce software components and applications.

Key words: Component-based programming, E-commerce, Object-oriented technology, WWW.

I. Introduction
The World Wide Web and the Internet have already changed the way that many companies conduct their business and the demand for electronic commerce software is growing exponentially [Hamilton 97]. Electronic commerce software must be very flexible and interoperable because business practices vary tremendously among companies and the need of exchanging services among companies and customers. To meet these requirements, a large number of small-scale development have resulted in a variety of solutions, mostly individual products of e-commerce [Kosior 97]. Because these narrowly limited products are not very desirable in e-commerce, small developers are recognizing that they need a standard, common framework. This paper tries to apply the component-based software development methodologies to develop an e-commerce component framework (ECF) for developing e-commerce systems.

E-commerce systems today encompass a wide variety of business processes as well as providing a basis for business-to-business interaction and business-to-customer interaction. As such e-commerce is not restricted to a single application, but to a collection of applications, with trading partners selectively integrate their applications using the Internet as the infrastructure to streamline business processes [Sharma 99]. The World Wide Web and the Internet provide a computational environment with pervasive use of interactive and high-bandwidth communication networks. From a system architect/developer's point of view, the building of a Web-based e-commerce system can be far more challenging as it includes comprehensive interaction between component-based system elements in various situations [Vetter 99]. The ever changing and evolving nature of business processes and e-commerce initiatives requires e-commerce applications to be flexible and extensible. Companies experienced with multiple e-commerce applications have learned that they need a unified, incremental approach capable of supporting and sustaining multiple and continuously evolving e-commerce initiatives.

A model that is specifically suited to these requirements is that of component-based software development. Presently businesses are faced with the dubious choice of buying packaged e-commerce software solutions and modifying them as much as possible to suit their business, or they can fund the development of a tailor made system from scratch. Both of these methods are undesirable. The rapid evolution of e-commerce initiatives means that tailored applications can be outdated before they are even completed. The cost of developing such a system is large considering this prospect. Such systems cannot be modified with ease, most require a new version to be created to include new features and innovations. In buying a generic solution the business must often deal with features that are near enough to what their business really requires, customizing pre-packaged applications can require longer time frames and cost from 50% to 100% more than original costs [Sharma 99]. Component technology could overcome many of these problems by providing the best of both worlds.

The aim of this paper is to investigate the theoretical and practical aspects of constructing a framework for building component-based electronic commerce systems.

Until now, component-based software development methodologies have remained largely within the circle of full-time programmers developing
system software or traditional application software. However, component-based software development methodologies are likely to expand toward an audience that is less technical and more application domain-oriented. It would be highly desirable to have integrated development and runtime environments that isolate much of the conceptual and technical complexity involved in building e-commerce applications. With the help of our proposed ECF, businesses can develop highly reusable e-commerce components and can carry out the relatively simpler tasks of assembly of e-commerce components developed in-house or purchased on the market.

2. The Architecture

E-commerce components are software entities that perform a limited set of tasks within an e-commerce application. Using industrial strength software components or pre-fabricated software building blocks that support the application, to construct e-commerce applications allows a company to select the features that their business requires, and develop a tailored application in a short period of time by using rapid application development and an assemble approach. The assemble approach lets developers mix and match the best of breed solutions to build mission critical e-commerce applications that are scalable, reliable, reusable, and interoperable. As we have surveyed in the previous section, the leading component-based software development methodologies are OMG's CORBA, Sun's JavaBeans and Microsoft's DCOM (Distributed Component Object Model) and ActiveX (Adler and Krieger 98, Goscinski and Zhou 99).

A software component is a unit of composition with contractually specified interfaces and explicit context dependencies only. A software component can be deployed independently and is subject to composition by third parties [Szypperski 98]. Every component must have a well-defined standard interface or set of standard interfaces and open Application Programming Interfaces (API) so it can be successfully used in a plug and play environment.

Our electronic commerce component framework is an environment that supports the development and reuse of plug and play e-commerce components, and the process of assembling them into portable, interoperable e-commerce systems. The e-commerce platform should standardise e-commerce component development, enable the developer to customize the functionality and appearance of components and facilitate simple application construction.

We have carried out an initial survey on the technologies used in ActiveX [Microsoft 98], JavaBeans [Thomas 00], EcoSystem [Tennenbaum et al. 97], project Offer [Bichler and Segev 98], and other related projects. Based on our initial work we designed a framework for building component-based software with the following four elements:

- Objects: Objects are the minimum building blocks of components viewed from the component-based programming point of view. Each object has a unique identity, a set of data structures that can be used to hold the state of the object, and a set of operations that can be used to manipulate the data stored in the object's data structure. An object cannot exist by its own as a useful program.

- Components: Components are independent software entities possess some program logic, ranging from small graphical user interface widgets such as buttons to complex components such as stock ticker display, to full-size applications such as word processors and spreadsheets.

- Containers: Containers are software entities used to assemble components. They provide a context in which components interact and may be arranged. Containers can be nested within other containers.

- Glues: Glues are scripting languages used to initiate and direct interactions between components and objects. They are used to describe the relationships among objects and components and to assemble objects and components into larger components or even applications within a container.

To successfully design the framework, the following issues have to be answered:

- How to define and describe objects, components, and containers?
- What are the relationships among these elements?
- How to prove that a particular glue of certain objects and components is correct?

In our previous research [Chen and Zhou 00] we have developed a modeling language based on the Real-time Object-Oriented Modelling (ROOM) language [Selic et al. 94] to model reactive system entities and their relationships. Since the reactive entities (sensors, actuators and decision making managers) and their relationships with other entities have many similarities to objects, components, and containers and their relationships, we applied some of our previous results to the understanding of the above issues. We discuss these issues in a separate paper.

3. Services for Component-Based Software

Our framework for building component-based electronic commerce systems includes the following essential services: component publishing and
discovery service, event handling service, and an assembly line.

- **Component publishing and discovery service**: When a component is created or is placed in a container, it needs to identify itself and the interfaces it supports. The new component registers or publishes its existence and interfaces with the framework. As a consequence, other components learn through the discovery facility of its existence and how to interact with it. The implementation of this service is an extension of the global name server that we have developed before for distributed objects [Wang and Zhou 99].

- **Event handling service**: Objects and components communicate with each other through messages. An object or a component raises or broadcasts a message or an event, and the framework is responsible for delivering the message to the appropriate objects or components. Messages may be generated by the system itself, for example, by a click of the mouse; or they may be generated by other objects, such as when a database record is changed. The implementation of the event handling service is based on our previous research for reactive systems [Chen, Zhou and Yang 99] and the sensors and actuators [Zhou and Eide 98] created for our reactive system.

- **Fault-tolerant service**: This service includes two classes:
  1. A generic Java fault-tolerant policy class that implements some fault-tolerant policies for replicated components/objects. In particular, the following replication-based fault-tolerant policies [Guerraoui and Schiper 97] and their variations are implemented [Wang and Zhou 99]:
    - Primary-backup replication policies. These policies use a primary replica to receive and respond to client requests. Other replicas form the backups and will be in warm standby or cold standby states. One of these backups will be switched in when the primary fails to function.
    - Active replication policies. These policies give all replicas the same role without a centralised control. All replicas are actively involved in services. If a replica fails to function, other active replicas will replace its services.
  2. A group communication class that offers mechanisms of reliable multicasting, ordered message delivery, failure atomicity, and consistent group membership view. These features are essential for the coordination among components in an e-commerce application. In particular, two levels of services are provided by this sub-system:
    - Reliable and orderly message delivery primitives. Message delivery primitives are supposed to be fault-tolerant with respect to site crash failure, message loss, and other detectable failures. By tolerating failures at primitive level, it reduces burden on application programmers, and also retains consistent failure reporting and handling.
    - Group view services. Mechanisms like **formGroup**, **joinGroup**, **leaveGroup**, and **destroyGroup** have been provided to create and to maintain a consistent group membership view between group members in the presence of failures.

- **An assembly line**: Components can be assembled during run-time, or during development period. In both cases, an assembly line is required to perform the job. The assembly line enables objects and components to expose their properties and behaviours to development tools or other components. The assembly line can provide mechanisms such as inspectors, editors, and debuggers for assembling components into applications. The assembly line is currently under development based on our design pattern for reliable distributed objects [Wang and Zhou 98].

4. **The Development of an E-commerce Component**

4.1. **Components of a Search Engine**

The development of a search engine component for an e-commerce system was undertaken to illustrate the value of the component architecture when designing and constructing e-commerce systems. The search engine itself is a component that is made up of many smaller components, class libraries, web pages and databases for example, that interact with one another to provide the required functionality.

It is proposed that the database component of the search engine will store information about websites, where users can submit their site to the search engine to publicise its existence to a wider audience. Users will be able enter keywords and the search engine matches the keywords with related sites in its database listing. In approaching the project, a standard design was arrived at that divided the search engine into several smaller components, as depicted in the figure below.

The Web based components, the **URL Submission interface** and the **Search engine interface**, of the search engine both access the same program - the search engine component. Both the **URL Submission**
interface and the Search Engine interface are HTML documents mounted on a web server. The URL Submission interface will allow the user to input a URL, title, author, keywords, as well as a short description of their site. The HTML documents search and index both have forms, with each form containing a number of form elements.

4.2. Interfaces

Standard interfaces have been developed for each component in C++. The interface of a component defines the components' access points. These points allow clients of a component, usually components themselves, to access the services provided by the component. Because the component and its clients are developed in mutual ignorance, it is the contract that forms a common ground for successful interaction.

Figure 2 depicts the user interface of the search engine interface.

**Search Engine**

---

Figure 2. The user interface of the search engine interface.

The user interface for the URL submission interface is depicted in Figure 3.

**Search Engine - URL Submission.**

---

Figure 3. User interface of the URL submission interface.

The standard interface defined for the client component is illustrated in the following C++ segment.

```cpp
#include <iostream>
#include <string>
#include <list>
#include "rmonapper.h"
#include "search.h"
#include "index.h"
#include "dbmswrapper.h"

int main()
{
    // Declare pointers to STL linked list
    std::list<string> *Keywords;
    // Declare pointers to components
    Search "SearchComponent;
```
The client or driver program is used to control and manage the individual components, it creates, destroys, and calls components where appropriate. It can be thought of as the glue that binds all the components together.

The CGI Wrapper is defined as follows. This component is a wrapper around the GNU cgicc-3.1.1 library, where the object methods getElement() and getSplitElement() are called they are translated to the lower level Cgicc API.

```cpp
#include "cgicc/Cgicc.h"
#include "cgi.cgcic.h"
#include <cctype>
using namespace cgicc;

#define CGI_H
#define CCGI_H
#define CGWRAPPER_H

private:
    Cgicc cgi;
    public:
    CGWRAPPER();
    ~CGWRAPPER();
    // returns a string value of a form element
    string getStartElement(string name);
    void insertElement(string name, string s1, string s2);
    void getSplitElement(string& name, string& s1, string& s2);
    void closeConnection();
    void closeDBI();
    void query(string SQL);
    void insert(string SQL);
    }
};

The DBMS wrapper is defined below. This component is another wrapper class, around an existing class library. It is used to open, connect, close databases, close connections, query and insert records. As before the top level interface is translated down to the lower level DB API within the implementation of the component.

```cpp
#include <iostream>
#include <string>
#include "CGDDBWRAPPER_H"
#define DBDDBWRAPPER_H
class DBDDBWrapper
{
    private:
        DBDDBWrapper(string filename, string database, string password);
    public:
        DBDDBWrapper();
        ~DBDDBWrapper();
        void open(string filename);
        void closeDBI();
        void closeConnection();
        void query(string SQL);
        void insert(string SQL);
    }
};
```
The Index component is defined as follows. This component is used to firstly validate data passed to it and to generate an SQL statement for inserting or updating a record. The SQL statement is then used by the DB component.

```cpp
#include <db.h>
#include INDEX.H

class Index
{
  private:
    string SQLStatement;

  public:
    Index();
    ~Index();
    void BuildRecord(const std::string & Keywords, const string & URL, const string & Title, const string & Description, const string & Author, bool Update);
    void AddRecord();
};

#endif INDEX_H
```

The Search component is defined below. As with the indexing component the searching component also validates data passed to it, and generates an SQL statement for querying the DB.

```cpp
#include <db.h>
#include SEARCH_H

class Search
{
  private:
    string SQLStatement;

  public:
    Search();
    ~Search();
    void BuildQuery(const std::string & Keywords, string Order, bool Matches); // void Query();
};

#endif SEARCH_H
```

The relationships of these components can be illustrated as in Figure 4.

![Figure 4: The relationships of components](image)

The purpose of the development of the component-based search engine for a component e-commerce application is to explore the value of using the component-based development methodology for E-commerce applications. The search engine component developed can be plugged into existing applications with minimal adjustment.

5. Remarks and Future Work

The development of an e-commerce component framework enables developers to focus on the functionality of the end product rather than labouring over the development and adaptation of each individual component. It will remove the need for lengthy in house development of proprietary e-commerce systems and will promote shorter development times. Compared with existing e-commerce frameworks, the ECF developed in this paper has a simpler architecture and a set of clear concepts. Also, reliability is a built-in feature, not an add-on.

The novelty of this work lies in the innovative approaches to the building of electronic commerce systems through a research on the techniques, notations, and frameworks of the component-based technology. The research on the technology of component-based systems has been extensive, but the research on the well-defined techniques, notations, and frameworks for developing component-based systems is still pending. Without a clear understanding of these techniques, notations, and frameworks, the component-based technology cannot be widely accepted by the software industry.

We are currently carrying out the following further research:

- The design and implementation of the assembly line. Many research issues are also associated with the development of the assembly line and will be investigated thoroughly. For example, what scripting language(s) should be used during the assembly process? How do we know an assembly is correct? If there exist different assembly approaches, how can we select the one that will result in the most efficient component or application?

- The design and implementation of a set of generic e-commerce components. We plan to develop a number of generic e-commerce components, including a transaction management component, a catalogue management component, a content management component, a Web site component, a customer support component, and a security management component. The development of all these components will be based on our previous work in developing Web-based applications and therefore can be built up relatively quickly [Zhong and Zhou 98, Zhou 99].
Acknowledgement

This work is partially sponsored by an Australian Research Council's Small Research Grant.

References


<http://www.haas.berkeley.edu/~city/ecc/>


Java Sensors and Their Applications

Wanlei Zhou and Eric Eide

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
wanlei@deakin.edu.au

Abstract. This paper presents the design, implementation, performance analysis and applications of Java sensors based on the reactive system model. A Java sensor constantly monitors the state of an entity. Other entities can subscribe to a Java sensor and therefore are notified whenever the monitored state changes. The main advantage of using sensors is the ability to separate policies from mechanisms, providing greater flexibility and modularity in system design and programming. The paper uses two examples to show the application of our Java sensors.

Keywords: Java language, Reactive systems, Fault-tolerant computing, Distributed database systems, Mobile computing.

1 Introduction

Java virtual machines, which are rapidly becoming available on every computing platform, provide a virtual, homogeneous platform for distributed and parallel computing on a global scale [Gosling, 1997; Munson and Dewan, 1997]. A Java application usually consists of many objects (may be developed using languages other than Java) distributed all over the network and therefore it is essential for the control part of the application to know the current states of these objects. Java sensors proposed in this paper can be used to monitor and report the states of distributed objects.

Our Java sensors are based on the reactive system concepts. Several systems, such as Meta [Wood and Marrullo, 1994], DisCo [Systa, html] [Systa, 1995] and STATEMENT [Harel et al., 1990], and languages, such as Reactive C [Boussinot, 1991] and Reactive Pascal [Quintero, 1996] that is based on the reactive system concepts have been developed recently. However, most of the research on reactive systems is concentrated on process control (such as controlling a robot). The aim of this paper is to apply the reactive system concepts in developing distributed applications using the Java language.

Figure 1 depicts the architecture of our reactive system model. In this model, sensors are attached to applications to obtain their states (or equivalently, to monitor some events about the applications). These events are sent to the decision making managers (DMMs). The DMM reacts to these events by using the actuators to change the states of the applications.

The major advantage of this model is the separation of the mechanism for gathering information and the policy for making decisions according to the information received, and as this can lead to a better software architecture.
Sensors and actuators can be simple or composite. A simple sensor (simple actuator) can only be directly attached to one application. Figure 2(a) shows a simple sensor (actuator). A composite sensor (composite actuator) can consist of multiple sensors (actuators) from multiple applications. Figure 2(b) shows a composite sensor (composite actuator).

The rest of the paper is organised as follows. Section 2 presents the implementation of our Java sensors. Section 3 discusses the performance issues of Java sensors. Section 4 presents the application of Java sensors in a replicated database system. Section 5 applies Java sensors in a mobile file system. Section 6 summarises the paper.

2 Implementation Issues

Only the implementation of simple sensors are dealt with in this paper. According to the way that a sensor performs its work, we have defined the following three types of simple sensors:

- Event sensor. It reports to its subscribers when the monitored event happens.
- Timer sensor. It periodically reports to its subscribers on the state of the monitored entity.
Polling sensor. It periodically checks some state of the application and reports to subscribers.

Using various features of the Java programming language, these sensors have been implemented as both single and multithreaded entities. Also, both multicast datagram and stream-based communication sensors have been implemented, including a hybrid approach using multicast datagrams on each local subnet, and stream-based communication between subnets.

2.1 Multicast Datagram Sensors

Multicast datagram sensors use multicast datagrams to communicate to subscribers. Using a single thread of control, a datagram is sent out onto the subnet using an address reserved for multicasting, whenever a specific event occurs. Other entities can subscribe to the sensor, simply by creating a multicast socket, and using the joingroup and leavegroup primitives provided in the Java language.

The three specific sensors implemented offer the following interface and services:

- Multicasting timing sensor. This sensor is implemented as a stand alone program, but can easily be embedded into another software entity. Each timing sensor inherits from the Java object thread.
  - **TimingSensor()**: the constructor to the sensor, creates a multicast datagram socket, establishes the multicasting and local host addresses and formats the datagram ready for transmission.
  - **run()**: this method emulates a timer by putting the thread of execution to sleep for some predefined period of time. Once this interval expires, the sensors transmits its datagram, which is multicast on the local subnet, and then is put back to sleep.

- Multicasting Polling Sensor. This sensor is implemented as a stand alone program, and inherits from the Java object thread.
  - **PollingSensor()**: constructor to the sensor, creates a multicast datagram socket, and establishes the multicasting and local host addresses. Formats packets to transmit when the polling results are available.
  - **run()**: this method also emulates a timer by putting the thread of execution to sleep for a predefined period of time. Once this interval expires, the sensor executes a predefined polling task (e.g., to check the states of a process or a file). Based on the result of this execution, an appropriate packet is transmitted onto the subnet.

- Multicasting Event Sensor. The implementation of this sensor requires it to be embedded within an application.
  - **EventSensor()**: constructor to the sensor, creates a multicast datagram socket, establishes the multicasting and local host addresses and formats the packet ready for transmission.
- Trigger(): public method used by the application to invoke the sensor to transmit a datagram, hence reporting to its subscribers an event has occurred. Invoked within the application by the call: `<sensor-name>.trigger()`.

In order to avoid packets being flooded onto other networks, Java multicast datagrams include a TTL (Time To Live) field in the header to reduce packets unnecessarily circulating and congesting networks. The programmer can increase the TTL allowing the packet to travel further.

However, many routers provide only limited support for multicasting. Packets are allowed to be multicast within a subnet, but not to pass through the router into other subnets. To overcome this, a tunnelling approach has been implemented.

Tunnelling (as shown in Figure 3) establishes two software entities, one located on the sensor’s subnet, and another on the remote subscriber’s subnet. On the sensor’s subnet, a courier joins the multicast group, and retransmits the locally multicast packets using either datagrams or sockets to another remote subnet. At the other end of the tunnel, a publisher receives these packets and multicasts them onto the remote subnet using the same address/port as the sensor.

![Diagram](image)

**Fig. 3. Tunnelling multicast packets to other subnets**

### 2.2 Stream Based Sensors

The second implementation of sensors use Java stream based communication to provide reliable reporting to subscribers. Using multiple threads, a message is sent out to each subscriber using dedicated connections established between the sensor and its subscribers. Each connection is handled by its own thread of execution. Other entities can subscribe to the sensor by requesting a connection,
which creates a new thread to handle that connection. In order to synchronise
the transmission of reports, each sensor creates a ThreadGroup, to which each
new thread created is placed into. Figure 4 shows the architecture of a stream
based sensor.

![Diagram of stream-based sensor architecture]

**Fig. 4. Stream-based sensor architecture**

The steam sensor offers the following interface and services:

```java
// Sensor stream sensor extends Thread {
    protected ThreadGroup
    protected Listener connection;
    protected outing message;
    protected Init report;
    // StreamSensor constructor, creates a ThreadGroup, one listener object, and
    // listener for one thread method;
    protected streamSensor that port ( ... )
    // Subclass needs to be implemented for each specific sensor
    public void name ( ... )
    // Invokes each thread within the ThreadGroup to send a report to their subscriber
    protected void report ( ... )
    // This ensures establishment a communication channel between the sensor and
    // the subscriber
    Class GroupMembers extends Thread {
        protected ThreadGroup
        protected Group Members
        protected ReportListener
        // Constructor of Group Members. adds itself to the ThreadGroup,
        // identifies the host machine address and create a message stream to send,
        protected Group Members (Thread Group, Group Member, String id) ( ... )
        // Invokes the Group Members' thread of execution. Then the process again on
        // report to event. It returns each Group Members' thread of execution, allowing
        // them to send their message to their subscribers. Then retransmit the
        // Group Members suspend their thread of execution, and so on.
        public void name ( ... )
    }
    // Listener class used for connection requests and creates a new thread to handle
    // such connection
    Class Listener extends Thread {
        protected Group Members
        protected Group Members
        protected Listener
        // Constructor of the listener, creates a Java socket to receive
        // connection requests from subscribers
        protected Listener (Thread Group members) ( ... )
        // Receive connection attempts from subscribers and create a new socket for
        // each subscriber. Create a Group Members object for each socket connection, passing
        // the key socket as a parameter.
        public void name ( ... )
    }
```

The three types of sensors inherit from the StreamSensor class:
Stream based timing sensor. Implemented as a stand-alone program, but can also be embedded within an application. The run() method is redefined by putting threads to sleep for a fixed period of time and then reports to the subscribers with parent method report().

Stream based polling sensor. Implemented as a stand-alone program and inherits from the StreamSensor class by changing the run() method to periodically execute a predefined task. Currently two such tasks are defined. The first is to check the liveness of a process, and the second is to check the states of a file.

Stream based event sensor. Implemented as a software entity that must be embedded within an application. A new method, called trigger(), is invoked by the application when some event (within the application) has occurred (using the call <sensor-name>.trigger()). Two tasks, similar to the polling sensor, and a task to report an empty event, are predefined.

3 Performance Issues

The following summary outlines a series of tests and results used to evaluate the performance of each type of sensor. The test site consisted of a collection of networked (10M Ethernet) Sun spare machines running the Solaris operating system. Sun’s JDK 1.1.2 Java interpreter is used to run both the sensors and the subscribers.

Three groups of tests have been performed, each in a different distribution environment. The first tests are conducted with the sensor and subscriber(s) located on the same host. The second group of tests is conducted with the sensor and subscriber(s) located on the same subnet but on different hosts. In tests with more than one subscriber, each subscriber is located on a separate host. The final group of tests is conducted with the sensor located on a remote subnet from the subscriber(s) subnet. As in the previous test group, in cases when there is more than one subscriber, each subscriber is located on a separate host.

3.1 Test Descriptions

The tests are conducted on two groups of three sensors, the first group using multicasting datagrams to report to subscribers and the second group using stream-based communication. Within each of these groups are a timing sensor, polling sensor and an event sensor.

The role of the timing sensor is to report to subscribers at discrete intervals of time. By setting the time required for sensing activity to 0 (i.e., no sensing activity is performed) tests show how long a sensor takes to send a message, without the overhead of the sensing activity.

The polling sensor tested is required to report on the liveness of a sleep process running in the background on the same host (of course, the sensing activity
can be any predefined task). The sensor determines whether the process is running and reports its state to the subscriber. These tests measure the overhead of checking the liveliness of a process and the time taken to report this liveliness to the subscriber.

The event sensor tested is embedded within a test application which triggers the sensor to report to its subscribers when an empty event has occurred. These tests measure the overhead of the sensor being triggered by an application and the time taken to report this event to the subscriber.

Each sensor is tested 1000 times with an average time taken from these results.

3.2 Results and Discussions

Table 1 shows the time needed by various types of sensors to report to a single subscriber. From this table, we can see that the multicasting sensors have shown an expected gradual increase in reporting time as the subscriber is located further and further away from the sensor. Once the subscribers are located on a subnet remote to the sensor’s subnet, an added overhead of tunnelling the datagram from one subnet to another is recorded.

<table>
<thead>
<tr>
<th></th>
<th>Timing</th>
<th>Event</th>
<th>Polling</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>LH</td>
<td>LS</td>
<td>RS</td>
</tr>
<tr>
<td>Multicasting</td>
<td>3.33</td>
<td>4.24</td>
<td>12.23</td>
</tr>
<tr>
<td>Stream-based</td>
<td>3.84</td>
<td>5.26</td>
<td>6.102</td>
</tr>
</tbody>
</table>

Table 1. Reporting time (single subscriber). LH: local host; LS: local subnet; RS: remote subnet

Multicast datagrams are an unreliable method of reporting events, and therefore may have limited application in distributed systems covering multiple subnets. However, within a local subnet, the multicasting sensors are the fastest to report events and are not affected by an increasing number of subscribers.

Stream based sensors also show a gradual increase in reporting time as the distance between the sensor and the subscriber increases. However, no significant increase in reporting time is recorded when the subscriber is moved from a local subnet to a remote subnet because there is no tunneling overhead. Based on this result, and the fact that stream based sensors sensors utilise reliable communication for reporting events, these sensors may be more suited to distributed systems than the multicasting sensors.

Throughout all the tests conducted, the polling sensor is slower to report to subscribers by approximately a factor of ten. These results can be attributed to the execution of a system call to determine if a process is alive on the local host. This delay may limit such a sensors application, such as in hard real-time systems.
The following results show the effect of multiple subscribers on the reporting times when stream-based timing and event sensors are used to report events to 1 through 10 subscribers. Figure 5 shows the results of local host testing (where all timing and event sensors and their subscribers are located on the same host) and local subnet testing (where all timing and event sensors and their subscribers are located on the same subnet).

![Graph](image)

**Fig. 5. Local host and local subnet testings** *(EventLH.dat: event sensors running on a local host; TimingLH.dat: timing sensors running on a local host; EventLS.dat: event sensors running on a local subnet; TimingLS.dat: timing sensors running on a local subnet)*

Figure 6 shows the result of local host testing and remote subnet testing (where all timing and event sensors and their subscribers are located on two interconnected subnets (about 100km apart)).

The two figures (Figure 5 and Figure 6) confirm our intuition that as the number of subscribers increases, the time used by sensors to report to their subscribers increases as well. However, such an increase of time is not very dramatic in cases of a local subnet and remote subnets. This is good news for us since...
most of the DMMs require reports from sensors located in different hosts. We can also see that the stream based event sensors take slightly more time to report to subscribers than their corresponding timing sensors. This is because that an event sensor has to be triggered within an application, while timing sensors fire their reports automatically.

4 Example 1: Fault Tolerance in a Replicated Database System

Our first application example is a replicated database system where two Mini SQL (mSQL) [Hughes, 1996] database servers (called replicas) are running on two workstations located about 100 km apart. All database servers store identical information initially and each of them can accept client requests that read or update stored information independently. The task of the replicated database system is to maintain the data consistency among all the replicas, even in the case of system component failures. Figure 7 shows the architecture of the replicated
In this architecture, there is a replication manager running on each host where an mSQL database server is running. A client connects to a replication manager in order to obtain database services. If a client requires a read-only operation, this request will be serviced by the local replication manager by reading from the local database server. If a client wants to perform an update operation, the operation has to be performed in all database servers.

However, some decisions have to be made in case of system component failures. Here is a list of such situations:

- Case 1: A database server fails. For example, we assume that DB1 on Computer 1 fails. In that case, RP1 on Computer 1 has to re-direct all requests to DB2 on Computer 2. If such a request is an update request, then RP1 on Computer 1 has to store such an update in a stable storage (e.g., disk) and has to perform it on the failed database DB1 when it recovers. Similarly, when a client issues an update operation through RP2, RP2 has to store that operation in a stable storage and has to perform it when DB1 recovers.

- Case 2: A replication manager fails. For example, we assume that RP1 on Computer 1 fails. In that case, all requests have to be submitted through RP2 on Computer 2.

- Case 3: Computer 1 (or Computer 2) fails. For example, we assume that Computer 1 fails. In that case, all servers running on Computer 1 fail. All requests have to be submitted to RP2 on Computer 2. If there is an update operation, it has to be recorded in the stable storage and has to be performed when Computer 1 recovers (and DB1 recovers as well).

Therefore it is essential for a replication manager to know if a database server is alive in Case 1 and Case 3. It is also essential for a client to know if a replication manager is alive in Case 2 and Case 3.

We have used our Java sensors to deal with database server failures (Case 1). In particular, the following two methods are used:
- Use polling sensors. A polling sensor is run on Computer 1 to report the liveness of database server DB1 and another polling sensor is run on Computer 2 to report the liveness of database server DB2. The DMM modules of both replication managers subscribe to both polling sensors and are informed of the liveness of the two database servers periodically.

- Use event sensors. An event sensor is embedded into RP1 to report the failure of the connection between RP1 and DB1. Similarly, an event sensor is embedded into RP2 to report the failure of the connection between RP2 and DB2. The DMM modules of both replication managers subscribe to both event sensors and are informed the liveness of the two database servers periodically.

In both cases, the DMM modules stay the same. Their main task is to make decisions (as outlined in Case 1) according to the reports from the sensors they have subscribed. This shows the advantage of separating the mechanism from the policy.

5 Example 2: File Consistency in Mobile Computing

Many researchers have addressed the file consistency problem in mobile computing and some toolkits, such as the Coda file system [Kistler and Satyanarayanan, 1992] and the Rover toolkit [Joseph et al., 1997] have been developed recently. These toolkits differ in the features they offer and the trade-offs they make between flexibility and ease of use. For an overview of some well known toolkits, including the above mentioned two toolkits, please refer to [Grudin, 1994].

This application example comes from a real life requirement of one of the authors. The author uses a Sun workstation at work and downloads some of the files (or even a whole sub-directory) from the Unix box into his laptop computer (running Windows NT operating system and its file system can be integrated into the Unix file system) to work at home or places other than the office. This constantly creates file consistency problems since there are two copies of some files that can be updated simultaneously. Also sometimes a sub-directory contains a few hundred files and it is a painful process to remember which of them have been changed.

We have then designed and implemented a program using our Java sensors (polling sensors in this case) to automatically monitor files in two (or more) locations (in particular, sub-directories located in Unix file system and the laptop computer). Any file inconsistency of the two monitored locations will be reported to the DMM module of the program (a Java program, can be executed on the Sun box or the laptop computer) that subscribes to the sensors.

An alternative implementation can use the event sensors to record any update operations on files located in monitored sub-directories. When the laptop computer is linked to the network, the DMM module of the program (which subscribes to the event sensors) can get a complete report on the updates carried out during the disconnection and then can act accordingly. However, in both cases, the DMM module will stay unchanged even when the mechanisms are different.
6 Summary

The design, implementation, performance analysis and application examples of our Java sensors are presented in this paper. The main advantage of Java sensors is the potential of achieving the separation of mechanisms and policies in software development, inherited from the reactive system concepts. The two application examples show the potential benefits of our Java sensors.

References


Building Reliable Programs in a Replicated Environment Using Remote Procedure Calls

Wanlei Zhou and Andrzej Goscinski
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
{wanlei, ang}@deakin.edu.au

Abstract This paper addresses the problem of building reliable computing programs over unreliable remote procedure calls (RPC) systems by using replication and transaction techniques. We first establish the computation model: the RPC transactions. Based on this RPC transaction model, we present the design of our system for managing RPC transactions in the replicated-server environment. Finally we present some results of a correctness study on the system and an application example of the system.

Keywords: Distributed computing, fault tolerance, replication, remote procedure call, transaction

1 Introduction

Remote Procedure Call (RPC) is perhaps the most popular model used in today's distributed software development and has become a de facto standard for distributed computing. Many leading computer companies have agreed on a vendor-neutral distributed computing environment (DCE) architecture proposed by the Open Software Foundation [5] that follows the RPC paradigm. Although the DCE architecture helps reduce the heterogeneity of server-access protocols, one important issue is still outstanding: the support for fault-tolerant services.

It has been suggested that the use of replication and transaction techniques can provide an environment for developing reliable programs [2]. There have been some efforts to combine two of the three techniques -- RPC, replication, and transaction -- together to achieve reliable computing. For example, the ISIS toolkit [7], the location-based paradigm for replication proposed by Tripodilou and Taylor [6], the RPC transaction management system proposed by Zhou and Molinas [10], and the Coda file system [3]. However, none of the existing systems/proposals are completely satisfactory.

The purpose of this project is to combine replication, transaction management and RPC together to form a reliable and efficient distributed computing environment. Fault-tolerant programs developed in our environment should be able to tolerate single failures such as a server failure, a site failure or even a network partition without involving manual intervention. These programs will also have efficiency similar to a non-replicated system.

2 System Models

A distributed system with replicated servers consists of many sites interconnected by a communication network. A service is provided by a group of replicated servers (called replicas) executing on some sites. These replicas manage some common data objects that can be shared by many clients.

For simplicity, we assume that each replica knows the location of other replicas that store the same data objects. This assumption can be loosened if a replication directory service is used.

We model a service as a set of replicas: \( S = \{S_1, S_2, \ldots, S_i, \ldots, S_n\} \), where \( i = 1, 2, \ldots, n \), are called the sequence numbers of these replicas. Each replica \( S_i \) manages a set of data objects: \( O^i = \{d_1^i, d_2^i, \ldots, d_m^i\} \). The consistency constraint requires that for \( i, j = 1, 2, \ldots, n \) and for each service \( S, O^i = O^j \).

Each replica in our system provides a number of remote procedures that can be called by clients for processing the data objects managed by the replica. We use \( P \) to denote the set of all remote procedures provided by all replicas of the system:

\[ P = \{ p \mid p \text{ is a remote procedure provided by the system} \} \]

We define the effects of an RPC as the processing of one data item of the object system. Hence we can abstract an RPC as a mapping of the following type

\[ c : P \times O \rightarrow \{OK, FL, US\} \]

where \( O \) is the union of all data objects managed by all services of the system. The values of the target set have the following meaning:

- **OK** This means that no failure occurred during the RPC's execution.
- **FL** This means accessing failure. This means that the RPC has not executed.
- **US** This means unknown state. By \( c(p, d) = US \), we mean that the client cannot tell if the RPC...
has or has not been executed because various reasons.

Without loss of generality we assume that all RPCs are update-oriented operations. That is, if \( c(p, d) \) is successful, it transforms the data object \( d \) from the existing state to a new state. Reference \([10]\) contains more detailed description of our RPC model.

We define a (parallel) RPC transaction as \( T = \{ c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k) \} \), where \( c_i(p_i, d_i) \) is an RPC, and \( p_i \in P \), \( d_i \in O \). The semantics of an RPC transaction is that after issuing the transaction, all \( c_i \) of \( T \) will be executed if no error occurs (commit or OK), or if any of them fails, all executed RPCs will be rolled back (abort or FL). An RPC \( c_i(p_i, d_i) \in T \) returns OK if and only if all replicas of \( d_i \) have successfully performed the procedure \( p_i \). Similarly, \( c_i(p_i, d_i) \) returns FL if and only if some of its replicas of \( d_i \) failed to execute the procedure \( p_i \) on \( d_i \).

In addition to the two normal states (commit and abort), we define a partial commit (PC) state. The real meaning behind the PC state is that a replica is down or the network is partitioned, but there are other replicas that can provide the same service. So the transaction has only performed RPCs on all replicas that are alive (can be reached now). For those replicas that are not reachable now, the transaction effect will be resolved when these replicas re-join the service (e.g., the failed replica recovers or the network re-united).

The execution of all \( c_i \) in \( T \) is in parallel. Sequentially executed transaction can be easily established from the parallel model. However, if operations within a transaction are to be executed sequentially, the serializability \([1]\) must be considered. In this paper, we only consider parallel RPC transactions. Figure 1 indicates the semantics of an RPC transaction.

![Figure 1: Semantics of the RPC transaction](image)

There are four classes of failures in a replicated-server environment executing RPC transactions: transaction abort, replica-is-down, site-is-down, and network partition.

- An RPC transaction \( T \) aborts if any of its RPC returns an FL (e.g., the requested data object is not free), or \( T \) was in a PC state and then the conflict-resolution process identifies that \( T \) is un-safe.

- A replica is down means that the replica is not accessible.

- A site is down means that the replica(s) that runs (run) on it is (are) down. We assume that each site has only one replica running on it. That reduces the problem into a replica-is-down failure.

- A network partition failure means that replicas of a service may belong to two disconnected partitions.

We use system failures to denote the latter three failures and we assume a single failure for system failures. We also use replica failure to mean that a replica is down or the site that the replica is running on is down. A partition means the network is partitioned into two disconnected sets and both parts have some replicas running on them. In both cases, we can divide the replicas into two parts. For simplicity, we only assume crash failures in this paper. Byzantine failures can also be dealt with when the number of replicas of a service more than three and a majority voting scheme is used.

3 Algorithms

We define a primary replica for a data object \( d \) as a replica that is the best (e.g., the nearest site to the RPC transaction manager (described below) that accepts the client request, however, the measure is left for individual applications) in performing an RPC \( c(p, d) \). Any replica can be chosen as the primary replica for a particular RPC.

3.1 RPC Transaction Management

Three system components are involved in processing a transaction submitted by a client.

- An RPC transaction manager (RTM) accepts a transaction \( T \) from the client. The RTM sends each RPC \( c_i(p_i, d_i) \in T \) to a primary replica of \( d_i \) and asks the primary replica to check if the RPC can be performed or not. We denote this operation as \( a(c_i(p_i, d_i)) \). The RTM then acts as a coordinator for managing the atomicity of \( T \) through the help of the primary replicas. Listing 1 describes the RTM algorithm.

- A primary replica accepts, from the RTM, an RPC \( c_i(p_i, d_i) \) and the request to check the executability of the RPC (the \( a(c_i(p_i, d_i)) \) operation). The primary replica sends the RPC to all replicas of data object \( d_i \) and asks all replicas (including itself) to check if they can execute the RPC (e.g., if \( d_i \) is free). We use \( b(c_i(p_i, d_i)) \) to represent this operation. The
primary replica acts as a coordinator for managing the RPC \( c_i(p_i, d_i) \) to be performed on all replicas (including itself). Section 3.2 describes the coordinating algorithm.

- Each replica of the data object \( d_i \) accepts, from the primary replica of \( d_i \), the RPC \( c_i(p_i, d_i) \) and the request to check the executability of the RPC (the \( b(c_i(p_i, d_i)) \) operation). The replica then co-operates with the primary replica by returning the executability check and performing the RPC when requested. Section 3.3 describes the cooperating algorithm.

Figure 2 depicts the model for RPC transaction processing.

![Figure 2: RPC transaction processing](image)

We define a function \( \text{locatePrimaryReplica}() \) which takes as input an RPC and returns the primary replica location of the RPC. This function also guarantees that if \( c_i(p_i, d_i), c_j(p_j, d_j) \in T \) and \( d_i = d_j \), then \( \text{locatePrimaryReplica}(c_i(p_i, d_i)) = \text{locatePrimaryReplica}(c_j(p_j, d_j)) \). That means that the two RPCs of the same transaction will use the same primary replica if they access the same data object.

The algorithm \( \text{manage rpc transaction}() \) of Listing 1 implements the RTM. The constant \( \text{MAXRPCS} \) defines the maximum number of RPCs allowed in an RPC transaction.

### Listing 1: Algorithm 1 - The RTM Algorithm

```c
// receive RPC requests, send to the primary replica \( c_i(p_i, d_i) \) of the RPC
function 
  \( \text{start}(\text{primaryAddress}[i], T, \text{primaryAddress}[j]) \) 
  \( \text{null} \)
end
```

3.2 The Coordinating Algorithm

When a primary replica receives an RPC request \( c_i(p_i, d_i) \) from a transaction manager, it uses the coordinating algorithm to maintain the consistency of all replicas in terms of the RPC. In this algorithm, the primary replica uses the 2PC protocol to ensure replication consistency. The coordinating algorithm is listed in Listing 2. We assume that \( S_j \) is the primary replica.

### Listing 2: Algorithm 2 - The Coordinating Algorithm

```c
// receive RPC \( c_i(p_i, d_i) \) from \( S_j \)
function 
  \( \text{locatePrimaryReplica}(\text{primaryAddress}[i], T, \text{primaryAddress}[j]) \) 
end
```

---

245
3.3 The Cooperating Algorithm

When a non-primary replica receives a request from a primary replica, it checks whether the request can be proceeded or not and acts accordingly.

We use a Need-To-Do (NTD) table to record the events of partially committed RPCs. The NTD table of each replica is kept in stable storage \(^4\). Therefore, information stored in the NTD table will not be affected by system failures. The NTD table structure is listed in Listing 3.

When the primary replica asks for a partial commit for an RPC, all replicas (including the primary replica) will record this event in its own NTD table as a new entry.

We associate with each data object \(d\) a lock \(d.lock\) and a partial commit flag \(d.pc\). The actual effect of \(b(c(p, d))\) on \(d\) is to check or change the values of \(d.lock\) and \(d.pc\). That is, \(d.lock = -1\) (locked), \(b(c(p, d))\) returns FL. If \(d.lock = d.pc = 0\) (free and not partially committed), \(b(c(p, d))\) returns OK and sets \(d.lock = -1\). If \(d.lock = 0\) \& \(d.pc = n > 0\) (free but partially committed), \(b(c(p, d))\) returns PC and sets \(d.lock = -1\) and \(d.pc = n + 1\). If the replica is down or the network is partitioned during the operation (i.e., the replica is unreachable), a US is returned by the RPC system. The cooperating algorithm is given in Listing 4.

\(^4\) See "Asynchronous Receive Procedure Handles for this RPC"

\(^5\) "deadlock"

\(^6\) "corrupt"

\(^7\) "false"

\(^8\) "true"

3.4 Conflict Resolution

Two algorithms are needed during the recovery of a system failure. When recovering from a replica failure, the recovering replica has to send a "re-uniting" message (it includes the sequence number of the recovering replica) to other allying replicas. This enables the allying replicas send outstanding NTD table entries to the recovering replicas by using the send_out_ntd() algorithm.

When recovering from a network partition, replicas of each part of the partition have to send a "re-uniting" message (it includes the sequence numbers of all replicas of the partition) to replicas of the other part. This enables the exchange of NTD tables among replicas in both parts by using the send_out_ntd() algorithm. Listing 5 describes the send_out_ntd() algorithm.

Once all replicas have received the NTD table from the other part, the conflict_resolution() algorithm is used by all replicas to resolve possible conflicts and to finalize the outstanding partially committed transactions.

We define a leader of an NTD table as the first partial commit entry for a data object \(d\). That is if \(t\) is a leader in an NTD table \(T\), then \(t.previous = \emptyset\). The conflict_resolution() algorithm uses three functions for processing NTD table entries led by a leader:

- The abort_all \((t_{\text{next}}, q)\) function aborts all RPCs led by the leader \(q\) of the NTD table \(t_{\text{next}}\).
- The commit_all \((t_{\text{next}}, q)\) function commits all RPCs led by the leader \(q\) of the NTD table \(t_{\text{next}}\). The NTD table \(t_{\text{next}}\) is the replica's own NTD table.
- The commit_all_received \((\ell_c, t)\) function commits all RPCs led by the leader \(t\) of the NTD table \(t_{\ell_c}\). The NTD table \(t_{\ell_c}\) is the received table.
Listing 6: The *Function

\[
\begin{align*}
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\end{align*}
\]

Listing 7: The *Function

\[
\begin{align*}
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\end{align*}
\]

Listing 8: The *Function

\[
\begin{align*}
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(h, t) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\end{align*}
\]

Listing 9: Algorithm 5 — The Conflict Resolution Algorithm

\[
\begin{align*}
\text{NDT}(t, q) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(t, q) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\text{NDT}(t, q) \quad &/ \text{commit all RPCs led by } q, \text{own NTD table} */ \\
\end{align*}
\]

4 Correctness

In this section we only list the assertions on the correctness of the system. A full analysis of the correctness can be found in [8].

Assertion 1: If a transaction returns OK, all its RPCs have been executed successfully.

Assertion 2: If a transaction returns FL, no RPCs of the transaction have been executed.

Assertion 3: The NTD table will not grow indefinitely and any entry of the table will be deleted eventually.

Assertion 4: After the conflict resolution, all outstanding PCAs will be either committed or aborted.

Assertion 5: If a transaction returns PC, the transaction will be notified an OK or an FL return in a finite time.

Assertion 6: After the recovery of a system failure, all data objects managed by the replicas will be in consistent states.

Assertion 7: The proposed system has a similar response time as a non-replicated system.

5 An Application Example

The system described in this paper has been used in our implementation of a loosely integrated heterogeneous database system [9]. Currently two database systems (Oracle and MitoSQL) are running on two separate Sun workstations located about 100 km apart. We name the locations that the database servers are running as LA and LB, respectively. A third database server, the Ingres database system, is to be added into the system soon. The trader, trading agents, and user application programs can be executed on any Sun workstations. Figure 3 depicts the architecture of our loosely integrated heterogeneous database system.

Figure 3: Architecture of the loosely integrated system

The trader is a key component of the system — the system cannot function once the trader fails. Therefore it is replicated on each location. An RPC transaction manager also runs on each location for managing RPC transactions initialised from the location.
We have carried out the following experiments. Since $L_A$ and $L_B$ are symmetric, the following experiments are also true for $L_B$.

1. No failures. All read-only operations (import) to the trader from $L_A$ are directed to the local trader in $L_A$ by the local RPC transaction manager. All update operations (export) to the trader are processed as an RPC transaction that has to go through both trader servers. Upon receiving an update operation, the RPC transaction manager at $L_A$ uses the local trader server as the primary replica and makes the RPC transaction call. When the primary replica returns an OK, the update operation is then successful and the user can be sure that both trader servers are to carry the update operation.

2. The trader server at $L_A$ fails. All read-only operations to the trader from $L_A$ are directed to the remote trader in $L_B$ by the local RPC transaction manager. All update operations to the trader from $L_A$ will return a UN, then the local RPC transaction manager changes the primary replica to the trader server in $L_B$ and re-submits the RPC transaction. This time a PC is returned and the user can be sure that the system is going to perform the update operation on the un-accessible replica in $L_A$ when it becomes accessible.

3. The RPC transaction manager at $L_A$ fails. In that case the user simply submits all operations to the remote RPC transaction manager located in $L_B$ (since the addresses of all RPC transaction managers are well-known).

4. The trader server and the RPC transaction manager at $L_A$ fail. Similar to (2), but a PC is returned from an RPC transaction and the user can be sure that the system is going to perform the update operation on the un-accessible replica in $L_A$ when it becomes accessible.

5. The trader server at $L_A$ and the RPC transaction manager at $L_B$ fail. Similar to (2).

6. The trader server at $L_B$ and the RPC transaction manager at $L_A$ fail. In that case the user submits all operations to the remote RPC transaction manager located in $L_B$. Then a similar situation of (2) will follow (only change $L_A$ to $L_B$).

6 Conclusion

A system for building reliable computing over an RPC system is described in this paper. The system combines the replication and transaction techniques together and embeds these techniques into the RPC system. The paper describes the models for replicas, RPCs, transactions, and the algorithms for managing transactions, replicas, and resolving conflicts during system recovery. Finally an informal correctness analysis is carried out and an illustration example and an application example are described in the paper.

References

Fault-Tolerant Servers for the RHODOS System

Wanlei Zhou and Andrzej Goscinski

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia

Providing reliable services is one of the primary goals in designing a distributed operating system. Nowadays, we have seen a trend in distributed operating system design to shift from large kernel architectures or even monolithic architectures to microkernel architectures supported by the client/server model. This means that a lot of services of an operating system originally provided by the monolithic kernel are moved out of the kernel, forming individual servers. It is then crucial to guarantee that these servers will provide reliable services. This article describes a design, based on a twin-servers model, of fault-tolerant servers for the microkernel-based RHODOS distributed operating system. A model that supports fault-tolerant services is designed. The performance of the model is simulated and analyzed. A design that implements the model in the RHODOS environment is also outlined. © 1997 by Elsevier Science Inc.

1. INTRODUCTION

One of the main goals in designing a distributed operating system is to provide reliable services for applications running on the underlying distributed system. A distributed operating system manages many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever-increasing dependency being placed on distributed operating systems, the number of applications requiring fault tolerance is likely to increase.

The design and understanding of fault-tolerant distributed operating systems is a very difficult task. We have to deal not only with all the complex problems of distributed operating systems when all the components are well (Goscinski, 1991), but also the more complex problems when some of the components fail.

Today we have seen a trend in distributed operating system design shift from large kernel architectures, or even monolithic architectures, to microkernel architectures supported by the client/server model. This means that a lot of services of an operating system originally provided by the monolithic kernel are moved out of the kernel, forming individual servers. It is then crucial to guarantee that these servers will provide reliable services.

This article describes a design, based on a twin-servers model, of fault-tolerant servers for the microkernel-based RHODOS distributed operating system (De Paoli et al., 1994). The fault-tolerant technique used is a variation of the recovery blocks method (Hecht and Hecht, 1986) and the distributed computing model used is the client/server model. In the twin-servers model, two servers, called twins, work together to provide a particular service. If one of the twins dies, the surviving twin will be responsible for providing continuous services.

The model design is aimed toward building microkernel-based fault-tolerant operating systems. It can also be used, however, in application areas in which requirements are less severe than in, for example, the aerospace field but in which continuous availability is required in the case of some components failures (Boari et al., 1988). The application areas could be, for example, supervisory and telecontrol systems, switching systems, process control, and data processing.

Software fault tolerance refers to the set of techniques for continuing service despite the presence, and even the manifestation, of faults in a program.
There are many techniques available for software fault-tolerance, such as the $N$-version programming (Avizienis, 1985) and the recovery blocks (Hecht and Hecht, 1986). In the $N$-version programming, $N$ ($N \geq 2$) independent and functionally equivalent versions of a program are used to process a critical computation. The results of these independent versions are compared (usually with a majority voting, if $N$ is odd) at the end of each computation, and a decision is made accordingly.

The recovery blocks method employs temporal redundancy and software standby sparing. Software is partitioned into several self-contained modules called recovery blocks. Each recovery block consists of a primary routine, which executes critical software function; one or more alternate routines, which performs the same function as the primary routine, and is invoked upon a failure is detected; and an acceptance test, which tests the output of the primary (and alternate, if the primary fails) routine after each execution. A variation of this method is used in this article.

The reminder of this article is organized as follows. In Section 2, we give an overview of the RHODOS distributed operating system. Section 3 presents the model for supporting fault-tolerant servers. Section 4 analyzes the performance of the model. Section 5 describes the architecture of a proposed implementation of the model. Section 6 reviews some related work. Finally, Section 7 concludes the article.

2. OVERVIEW OF THE RHODOS SYSTEM

The RHODOS system is a modern, high performance distributed operating system based on the concept of a microkernel and the client/server model (De Paoli et al., 1994). Its architecture is very clear: on each host there is a microkernel, known as the Nucleus, which provides a consistent and abstract view of the underlying hardware to the higher level servers and processes. It provides support for local interprocess communication, basic memory management, process dispatching, general data and statistic collection, and interrupt and exception handling.

On top of the Nucleus there are two groups of servers: the kernel servers (also called managers) and the system servers, following the need for separation of policy and mechanism. The kernel servers have a close interaction with the Nucleus. They provide the functionality that is usually placed in a monolithic kernel and thus, are trusted entities that utilise a set of privileged system calls to manipulate microkernel data. They are responsible for the management of the basic logical resources of a distributed system: processes and processor time, remote communication, memory, devices and network delivery protocols.

The kernel servers include the Interprocess Communication Manager, Network Manager, Process Manager, Memory Manager, Migration Manager, Data Collection Manager, and Device Manager.

In order to support users in a friendly, efficient, and secure manner, the RHODOS distributed operating system provides services through system servers. The following RHODOS system servers have been developed (De Paoli et al., 1994).

1. Name Server: supports attributed naming and provides evaluation of user names onto system names;
2. Trading Server: provides user autonomy and object sharing between homogeneous systems through object export, import, and withdrawal;
3. File Server: provides disk, file, transaction and replication services and storage for workstations without their own external storage;
4. Global Scheduler: improves computational performance through sharing idle or lightly loaded workstations, by employing load balancing and static allocation;
5. Authentication Server: authenticates users and servers by supporting one-way, two-way, and conference authentication.

The primary interface between processes and the Nucleus is via the library/system called boundary. Because the Nucleus does not provide a great deal of services itself, there is only a limited number of system calls provided. Further services must be requested from the kernel and system servers via message passing or remote procedure calls. Figure 1 depicts the architecture of the RHODOS system.

As we can see from the overview of the RHODOS distributed operating system, the servers in the system play a very important role. If a kernel server or a system server fails, it will have a severe impact on the whole system. For example, if the Global Scheduler fails, no load balancing and static allocation will be performed and the computational performance of the system will be greatly degraded.

To tolerate server failures, it is necessary to build fault-tolerant servers. Our primary goal here is to provide continuous services in the case of a server or even a host failure, without or with little impact on the whole distributed system.
3. THE FAULT-TOLERANT SERVICE MODEL

3.1. The Client/Service Paradigm

To make our fault-tolerant service model more general, we use the client/service paradigm to reflect the client/server computing in the RHODOS system. The client/service paradigm is a higher abstraction of the client/server model. It provides a more user-friendly interface in the sense that it hides the server details from the user. In the client/server model, a client deals directly with a server. For example, a client locates a server by the name of the server, and then it is given the address of the server, whereas, in a client/server model, a client deals with a service (which is provided by a group of servers) instead. For example, a client locates a service by the service's attribute name, and then it is given one of the addresses of servers which offer the same service. In the most general sense, the client/service paradigm is a way of structuring distributed programs. It consists of the following three entities (Svobodova, 1985).

- **Service**: A service is a software entity that runs on one or more machines. It provides an abstraction of a set of well-defined operations.
- **Server**: A server is an instance of a particular service running on a single machine.
- **Client**: A client is a software entity that exploits services provided by servers. A client can but does not have to interface directly with a human user.

Ideally, as in the RHODOS system, when requesting a service, a client specifies the nature of the service through an attributed name. The exact location of the service and the server that provides such a service will be determined by the system for transparency reasons. The drawback of this approach is the delay incurred by the name resolution and evaluation facility. Thus, in the RHODOS system, some servers have well-known addresses and clients that (usually are also system processes or agents of system processes) can communicate with these servers directly through the well-known addresses.

Two approaches of providing fault-tolerant services in the client/service paradigm can be explored. The first approach is to look at the problem through the clients' viewpoint, that is, to make the clients fault-tolerant. The clients can continue their work by seeking an alternative service should the previous service request fail. The second approach is to look at the problem through the services' viewpoint, that is, to make the service fault-tolerant. The service consists of multiple servers that can provide alternative services to a client should some servers fail. Here we use the second approach and assume that the only failure points are the server failures.

3.2. Server Types

We can divide services provided by a distributed system into two categories: the processing-oriented services and the data-oriented services. A processing-oriented service, developed as a stateless service, provides some processing power to clients. It takes a client's request and processes it. No internal states of the service are changed because of the client's request. Such services include, for instance, the computing service, the printing service, and the compiler.
service. The most important characteristic of these services is that, if two or more servers provide the same service, these servers do not need to communicate with each other to maintain a consistent state. We name client requests to these services as processing-oriented service calls.

A data-oriented service manages one or more objects and clients access these objects by issuing some requests. The states of an object can be changed because of a client’s request. Such services include, for instance, the database service, the file service, the authentication service, and the directory service. For these services, it is critical to maintain a consistent object state among all the servers providing the same service. We name client requests to these services as data-oriented service calls.

A service can be both a processing-oriented service and a data-oriented service. In that case, it supports both processing-oriented service calls and data-oriented service calls. In terms of implementation, a processing-oriented service call can be implemented by either simple send() / recv() pairs if the message-passing method is used, or a simple remote procedure call (RPC) if the RPC method is used. A data-oriented service call, however, must guarantee the consistency of the data objects. It has to be implemented by either nested send() / recv() pairs if the message-passing method is used, or as a nested RPC call, or an atomic RPC transaction (Zhou, 1991) if the RPC method is used.

3.3. Fault-Tolerant Service Design

For each service, two servers are maintained to provide the fault-tolerant service. We refer to them as twins and denote them as \( T^1, T^2 \). These two servers usually (but not necessarily) locate on different hosts for increasing reliability.

When a client requests a service, the request (either a processing-oriented or a data-oriented service call) is mapped to one of the twins. Without loss of generality, suppose that twin server \( T^1 \) is selected. Thus, the request is sent to \( T^1 \). At the same time, a copy of the request is also sent to \( T^2 \) and is kept in its reserve. If \( T^1 \) has completed the service of the request, it notifies \( T^2 \), and \( T^2 \) will drop the copy of the request from its reserve.

At any time, if one of the twins, say \( T^1 \), dies for some reason, the surviving twin \( T^2 \) will continue the service agreed by \( T^1 \) by using the requests kept in its reserve. Then \( T^2 \) will invoke another server to be its new twin (also named as \( T' \)). After that, the normal operation of the twins is resumed. Clearly, this twin-servers model can survive single server failures.

Figure 2 depicts the design model. In the diagram, the client makes a service \( i \) request to one of the twins, say, \( T^1 \) (line a). This request is treated as a normal request by \( T^1 \)—it is serviced once \( T^1 \) is free. At the same time, a copy of the request is also sent to \( T^2 \), the twin of \( T^1 \) (line b). This copy is stored in \( T^2 \)'s reserve. Line c denotes the interaction between the twins. The interaction includes notification of the completion of a request and, if the request is a data-oriented service call, the calls for maintaining consistency of data objects.

Similarly, the client can make a service \( i \) request to \( T^2 \) instead of \( T^1 \), and a copy of this request will be sent to \( T^1 \)'s reserve. In that case, we have a solid line going into \( T^2 \)'s normal queue and a dashed line going into \( T^1 \)'s reserve. The interaction between the twins will point from \( T^2 \) to \( T^1 \)'s normal queue.

Figure 2. The fault-tolerant service model.
This demonstrates an important property of the twin model: the two twins are identical and symmetric.

Obviously, a process pair model, like our twins model, can only tolerate fail-stop or crash type of failures. Tolerating other types of failures (for example, the Byzantine failures) normally requires a majority voting, and that requires at least three processes.

4. PERFORMANCE EVALUATION

Obviously, our model can tolerate single-point failures: if one of the twins dies, the surviving twin will continuously provide services agreed by the failed twin. This is clearly an advantage compared with the normal single server model. However, every fault-tolerant method employs redundancy, and redundancy may imply increasing costs and/or decreasing performance. This section tries to evaluate the performance of our fault-tolerant model in various circumstances.

4.1. The Queueing Model and Its Simplification

The fault-tolerant model shown in Figure 2 can be viewed as a queueing model. Each twin has a normal queue and a reserve queue. Members of the reserve queue will be added to the end of the normal queue once the other twin dies. This can be treated as a bulk arrival to the normal queue. The probability of the bulk arrival is identical to the probability of a twin failure, and the mean size of the bulk arrival is the same as the mean number of requests (requests waiting in the normal queue and the one in processing) in a twin. The normal queue also has a feedback input from its twin, indicating the interaction between the twins.

We know that the two twins are identical and are symmetric. We can then simplify the model into a single service node with a feedback and bulk arrivals. This model is shown in Figure 3. Its parameters are as follows:

\( \lambda \): the arrival rate of the normal requests.
\( \mu \): the mean service rate of the server.

- \( p \): the feedback probability. It consists of the probabilities of the following two actions:
  - notification calls: when a twin completes a service to a request, it has to notify the other twin; therefore, the other twin can delete the corresponding request from its reserve;
  - consistency calls: when a request is a data-oriented service call, the data consistency must be maintained through certain mechanisms (for instance, the nested `send()` / `receive()` calls or the RPC transactions).
- \( X \): the average size of the bulk arrival. This is the same as the mean number of normal requests in a twin.
- \( q \): the probability of the bulk arrival. This is the same as the probability of a twin failure.

We are interested in the changes in the performance, measured in the response time \( R \), when the following parameters change.

1. The probability of a twin failure (\( q \)).
2. The probability of feedback (\( p \)).
3. The traffic intensity (\( \mu = \lambda / \mu \)).

Because the random processes of the model are not all independent (for instance, the average size of the bulk arrival \( X \) depends on the mean length of the queue), the model cannot be solved by using existing analytic models. Actually, the model can only be solved analytically if we assume that all the correlated random processes are independent. However, these assumptions are not realistic. For example, the bulk arrival \( X \) cannot be independent from the mean length of the queue, simply because once a twin fails; the other twin will put all the requests of the reserve queue into the normal queue. A simulation model is then built and some simulation is carried out.

Our simulation tool is based on MacDougall's `nspl` (MacDougall, 1987). We have the following assumptions when building our simulation:

1. Similar to most of the computer system performance evaluation processes (Kant, 1992), we assume that the arrival rate of normal requests \( \lambda \)
and the mean service rate $\mu$ follow independent exponential distribution;
2. The twin failure probability ($q$) and the feedback probability $p$ follow a uniform distribution (within (0, 1));
3. The average size of a bulk arrival is the same as the current length of the normal queue;
4. The time used by a twin for moving the requests from its reserve into its normal queue is zero. Also, the time used for invoking a new twin is zero.

The final assumption is only for easier programming and can be adjusted accordingly if needed. Notice that the feedback probability $p$ of a processor-oriented service can be very small (close to 0) if grouping and piggybacking are used to reduce the number of notification calls (see Section 5.2 for implementation consideration). However, for a data-oriented service $p \geq 0.5$.

In order to compare our twin-servers model with a single server model, we also build a queuing model for the single server model. The single server model can be viewed as an M/M/1 queue, with arrival rate of requests $\lambda$ and the mean service rate $\mu$. It has no fault-tolerant features. That is, if the server dies, all the client requests will be lost. Because it has no redundancy, however, intuitively it will have less overhead, and thus a better response time than our twin-servers model. We only show the results of the M/M/1 model in Figure 7 (it shows the response times when the utilisation rate changes). In other cases (Figures 4–6) the results for the M/M/1 model are straight lines when $p$ and/or $q$ change. Next we are going to find out what is the overhead (the time used for the fault-tolerant purpose only) for our model.

4.2. Simulation Results
The first result describes the relationship between the system response time and the twin failure probability. Figure 4 depicts the response time (the vertical axis, in time units) when the twin failure probability $q$ (the horizontal axis) changes from 0 to 0.05. The feedback probability $p$ stays at 0.5 and the traffic intensity is $u = 0.33$ (the mean service rate $\mu$ is assumed to be 1 time unit).

For the single server model, it is easy to know that the average response time will be $R = 1.49$ time units, no matter how $q$ and $p$ change.

We have the following observations from Figure 4.

- If $q$ increases at a lower range (for instance, from 0 through to 0.03 in our case), the response time will not increase too much.
- If $q$ stays in a lower range, it means that the twins are not likely to fail. In that case, the requests in a twin’s reserve will not likely be added into its normal queue. Therefore, the response time of the system will not increase too much.

- If $q$ increases at a higher range (for instance, from 0.03 through to 0.05), the response time will increase drastically.
- If $q$ stays in a higher range, it means that the twins are likely to fail. In that case, the surviving twin will have to frequently add the requests of its reserve into its normal queue. The size of the normal queue will increase. As the size of the reserve is the same as the size of the normal queue, it implies that the number of requests added into the normal queue will drastically increase when $q$ increases. Therefore, the response time will increase quickly.

![Figure 4](image)

Figure 4. The response time when $q$ changes ($p = 0.5$, $u = 0.33$).
The conclusion is that our fault-tolerant model does not impose too much overhead on the system when the twin failure probability is low.

The second simulation result describes the relationship between the response time and the feedback probability. Figure 5 depicts the response time (the vertical axis, in time units) when the feedback probability $p$ (the horizontal axis) changes from 0 to 0.6. The twin failure probability $q$ stays at 0.02 and the traffic intensity is $u = 0.33$.

We have the following observations from Figure 5:

- The response time will not increase too much if the feedback probability stays below 0.5.
- If only less than half of the requests are data-oriented service calls and the others are processing-oriented service calls, the twins will use less time for inter-twin communication. Therefore, the response time will not increase too much because most of the time of the twins will be used for processing normal requests.
- The response time will increase drastically if the feedback probability is over 0.5.
- If more than half of the requests are data-oriented service calls, the twins will have to spend most of their time on inter-twin communication. Therefore, the time used for processing normal requests will be less. When the requests processing rate of the twins is less than the rate of incoming requests, the response time will increase drastically and eventually the whole system will be blocked.

Of course, the traffic intensity $u$ also plays an important role in here. We will analyze its influence later.

The conclusion is that our fault-tolerant computing model will not work properly if most of the data-oriented service calls, and the activity is relatively high.

The simulation result relates the response changes in both the twin failure probability and the feedback probability. Figure 6 depicts the response time (the $z$ axis, in time units) when the twin failure probability $q$ (the $x$ axis) changes from 0 to 0.05. The traffic intensity stays constant at $u = 0.33$.

Figure 6 confirms our above observations. That is, if $q$ and $p$ stay low, our model will not impose overload on the system. We can also see additional properties. For example, if $q$ is high but $p$ (not $q$) is low, the system overhead will also be low. The traffic intensity will be decreased only slightly when $q$ is high.

The simulation result shows the relationship between response time and the traffic intensity. $q$ and $p$ are fixed at this time. Figure 7 depicts the response time (the vertical axis, in time units) when the traffic intensity $u$ (the horizontal axis) changes from 0 to 0.85. The dotted line shows the case of the single server model. The middle line shows the case when the twin failure probability $q$ stays at 0.01 and the feedback probability $p$ is 0.36.

The following observations from Figure 7:

- The traffic intensity increases within a lower limit, from 0.1 through 0.4 in our response time will not increase too much, though both $q$ and $p$ are relatively
• If the traffic intensity increase in the range (for instance, from 0.45 to 0.6), the response time will increase. Both $q$ and $p$ are relatively high. But time will only increase slightly if $l$ are relatively low.

• If the traffic intensity increase in the high range (for instance, higher $d$ the response time will increase drastically). But $q$ and $p$ are relatively low.

The conclusion is that if both relatively low, our model will not result in much overhead on the systemper service has a less incoming request rateprocessing rate (i.e., the traffic flow), converting the service into our will cause less overhead to the system even if both $q$ and $p$ are relatively high. If a service has a very high incoming rate, its processing rate is low (that means already congested without introducing the fault-tolerant features), converting the service into our twin model will cause much overhead to the system performance.

5. THE SYSTEM DESIGN

This section describes a design that implements the fault-tolerant service outlined in Section 3.3. A prototype of the design (with some features still under development) is now running on a network of SUN workstations using the microkernel-based RHODOS distributed operating system.

5.1. The Architecture of the Fault-Tolerant Service Design

Figure 8 depicts the implementation architecture. On each host, there is a local name server. It manages, among other things, the service locations and service-server mappings on its host. Generally speak-
ing, any service running on a host must be registered to the local name server. If that service is to be shared by clients other than the clients of the local host, it has to be registered to the global name server as well. The global name server is responsible for managing service locations and service-server mappings of the whole network. There should be at least one global name server on the network concerned. In the RHODOS system, however, kernel and system servers are not required to register to the name servers. The communication ports of many such servers are well-known ports and clients can make requests to these servers directly.

The communication between different system entities is provided by message passing (blocked or unblocked) or Remote Procedure Call primitives supported by the RHODOS Reliable Datagram Protocol, which is an efficient and fast transport protocol using the services of IP and Ethernet for communication in the RHODOS environment.

In the client/server model, a distributed program consists of a client program part and a server program part. In our system, a client program consists of a Client Driver and a Client Stub. The client driver controls the client's action, and the communication details are hidden in the client stub. The server program consists of a Server Driver, a Server Stub, a Procedures part and a Reserve. Similar to the client program, the server driver controls the server's action, and the communication details are hidden in the server stub. The procedures part realizes the real service operations. The reserve keeps track of the requests sent to its twin. The advantage of dividing programs into drivers and stubs is that the stubs can be automatically generated by stub generators (Zhou, 1994).

The communication between various system entities of the model can be described as follows. When the client issues a service request, the request goes to the local name server first (line a). The local name server checks to see if the service can be honored in the local host. If it cannot be honored in the local host, the local name server will ask the global name serve to map the service request (line b). If the global name server finds a suitable server, say $T^1$, the local name server will establish a mapping between the service request and the remote server $T^1$ on behalf of the global name server and

![Figure 8. The architecture of the fault-tolerant service design.](image-url)
the request is sent to \( T^1 \) (line c and d). At the same time, a copy of the request will be sent to the twin server \( T^2 \) for its reserve (line d').

If the service request can be honored in the local host (i.e., one of the twins, say \( T^1 \) again, resides on the same host as the client does), a mapping between the service request and \( T^1 \) is established, and the request is sent to the local server \( T^1 \). At the same time, the local name server will locate the other twin \( T^2 \), possibly through the global name server, and a copy of the request will be sent to \( T^2 \) for its reserve (line d').

The client's service request fails if both the local and the global name servers cannot complete the mapping. After the mapping is established, the client can then send requests to the twins directly (line c, line d, and line d').

If the twins' addresses are well known, the client can send the request and its copy directly (with the help of the IPC manager) to the selected server and its twin (line c, line d, and line d').

Because both twins' addresses are made available to the client, the client can randomly choose any one of the twins as the server to perform the request, and the other to keep the copy of the request. Statistically speaking, requests to the twins are evenly distributed and therefore load of requests are balanced.

If the selected twin server \( T^1 \) dies, its twin \( T^2 \) will transfer all the requests in its reserve into its normal request queue and execute them as if they were requests made to itself. At the same time, \( T^2 \) will invoke a new twin server to be its new \( T^1 \). The twins make regular checks to each other, making sure that the other twin is alive.

Before the services exported by the twins can be accessed by clients, the twins must register to the global (and local) name servers if the twins do not have well-known addresses. Lines f and g show the global registration.

Because the local name server and the global name server keep the critical information of services, they must be fault-tolerant themselves. So these name servers are also designed to be twins. For a particular name server, the location of its twin is a well-known address. And for all the local name servers, the location of the global name server is a well-known address.

5.2. Implementation Issues

The first issue that we have to consider is the naming of twins. In RHODOS all objects are identified by System Names (SNames), which combine facilities for location transparency, identification, protection, and accounting of objects into one data structure. The C data structure for SNames is shown below.

```c
typedef struct {
    unit16_t sn_signature;    /* The user's id */
    unit8_t  sn_type;         /* The object's type */
    unit8_t  sn_copy;         /* Which object copy */
    unit32_t sn_origin;       /* The object's origin */
    unit32_t sn_object;       /* The object's name */
    unit32_t sn_access_rights; /* The access rights */
    unit32_t sn_checksum;     /* 32 (64)-bit checksum */
} SNAME;
```

The main elements of interest to interprocess communication in this structure include the \( sn\_origin \), \( sn\_object \), and \( sn\_type \), which when combined can uniquely identify an object over an entire distributed system.

In our design, each server (actually, the communication port of a server) is assigned an SName when it is created (for example, \( twin\_1 \) and \( twin\_2 \) for the twins of a service). Each service is also assigned an SName (for example, \( service\_name \)) when both of its twin servers are available. Thus, after the registration (lines f and g of Figure 8) a service has an entry in the relevant name servers as follows.

```c
typedef struct {
    SNAME *service_name;       /* service's SName e4 */
    SNAME *twin_1;              /* the first twin's SName */
    SNAME *twin_2;              /* the second twin's SName */
} SENTRY;
```
The service’s SName service_name is made available to clients. When a client calls a service through its SName, the name servers will be responsible for providing the twin servers SNames of the service. After that, the client can call both twins through the IPC Manager.

If a client knows the SName of a server, it can call the server directly through the help of the IPC Manager. This is the case when a service has a well-known address (that is, the SENTRY entry is well known).

If a twin dies, the surviving twin will invoke a new server to be its twin. The new server may have a different SName from the failed one. Thus, it has to be registered to the relevant name servers, and the twin’s SName will be updated accordingly.

The second issue that we need to address is interprocess communication. The RHODOS system provides three interprocess communication primitives: send(), recv(), and call(). The send() and recv() primitives are the basic message passing primitives, and the call() primitive, which is logically composed of a send() and a recv(), together with a recv() and a send(), provide an RPC communication service. The send() and recv() primitives are used in our prototype implementation. Following is a more detailed description of these two primitives.

- **send(dest_addr, ret_addr, send_args, send_results)**
  It sends a message to another process, given the destination address (dest_addr) and where a return message should be delivered (ret_addr). Details of where the message resides and how many bytes should be sent are given in the third parameter send_args. The fourth parameter, send_results, indicates the results of the send() primitive.

- **recv(dest_addr, ori_addr, recv_args, recv_results)**
  It receives a message that has been delivered, given the destination address (dest_addr) from where the message is to be received and where the expected message has originated from the origin address (ori_addr). The third parameter, recv_args, provides details of the message, such as where the message is supposed to be placed and the length of the received message. The fourth parameter, recv_results, indicates the results of the recv() primitive.

All the addresses of the above primitives (dest_addr and ret_addr of send() and dest_addr and ori_addr of recv()) are SNames. They can be obtained through the help of the name server(s).

Figure 9 depicts the interprocess communications between the client and the twins when the SNames of the twins have been known.

For a process-oriented service call, which is shown in Figure 9a, the client first sends a normal request to one of the twins, say, T₁, and at the same time, a copy of the request is also sent to another twin T₂. After the selected twin T₁ completes the work, it sends back the results to the client and, at the same time, sends a message to T₂ asking it to drop the corresponding copy request from its reserve.

For a data-oriented service call, which is shown in Figure 9b, the interprocess communication is almost the same as the case of a process-oriented service call, except that after the selected twin T₁ completes the work, it also asks the twin T₂ to perform the work on its managed data object for achieving data consistency. Because T₂ already has the copy request, it can perform the work easily by moving the copy request from the reserve to its normal queue.

Figure 9c shows the interprocess communication when the selected twin T₁ dies. In that case, the surviving twin T₂ will perform the work by moving the copy request from its reserve to its normal queue. The work of invoking a new twin is not shown.

It is obvious that the data-oriented service calls are the most time-consuming calls. Various methods can be used to reduce the execution time of a data-oriented service call. Our design here uses the general approach, that is, a notification call is sent from the first twin to the second, and both twins perform the task. This is fine if the execution time for such a call is short. However, if the execution time is long, one can save some time by using only the first twin to execute the request and then sending the result to the second twin for updating only.

The third implementation issue is the detection of twin failures. As we mentioned in Section 5.1, twins make regular checks to each other, making sure that the other twin is alive. Currently, the checking interval (i.e., the maximum delay of detection) is designed at 180 seconds, and it can be easily adjusted to suit different applications. The overhead of failure detection is small because each checking RPC call (with a few bits of message) only uses about 3 ms on a lightly loaded network (Zhou 1995).

The checking message is also used to carry piggybacked notification messages from process-oriented services. Normally, all notification messages (i.e., messages asking the other twin to drop the dupli-
6. RELATED WORK

There have been many successful distributed computing systems using the client/server model. But few of them consider fault tolerance in their design, or they rely on users to build up the fault-tolerant features.

Many leading computer companies have agreed on a vendor-neutral distributed computing environment (DCE) architecture proposed on the Open Software Foundation (OSF, 1990). This architecture is designed under the client/server model and requires that the interactions between its components follow the RPC paradigm. Although the DCE architecture helps to reduce the heterogeneity of server access protocols, one important issue is still outstanding: the support for fault-tolerant services.

Notable works on incorporating fault tolerance features into client/server distributed computing are the Argus (Liskov, 1988), the ISIS (Birman and Joseph, 1987) (Birman et al., 1991), the Cygnus (Chang and Ravishankar, 1991), and an atomic RPC system on ZMOB (Lin and Gannon, 1985). The Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. Atomic transactions are serializable and indivisible. A user can also define some atomic objects, such as atomic arrays and atomic record, to provide the additional support needed for atomicity. All the user fault-tolerance requirements must be specified in the Argus language.

![Figure 9. IPC when implementing a process-oriented service call.](image-url)
The ISIS toolkit is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. A special process group, called fault-tolerant process group, is established when a group of processes (servers and clients) are cooperating to perform a distributed computation. Processes in this group can monitor one another and can then take actions based on failures, recoveries, or changes in the status of group members. A collection of reliable multicast protocols are used in ISIS to provide failure atomicity and message ordering.

The Cygnus distributed system is an instance of the client/service model and has some common features as our model. But it tries to solve the problem through the clients’ viewpoint, by making the client resilient to server failures. There are no facilities in the Cygnus system to allow the servers to provide reliable services.

The atomic RPC system implemented on ZMOB uses sequence numbers and calling paths to control the concurrency and atomicity and uses checkpointing to maintain the ability of recovering from failures. Users do not have to provide synchronization and recovery themselves; they only need to specify if atomicity is desired. This frees them from managing much complexity.

But when a server (or a guardian in the Argus) fails to function well, an atomic transaction or an atomic RPC has to be aborted in these systems. This is a violation of our continuous computation requirement. The client program of the Cygnus distributed system can survive from server failures, but it does not provide fault tolerance on the server’s side. The fault-tolerant process groups of the ISIS can cope with process (both clients and servers) failures and can maintain continuous computation, but the ISIS toolkit is big and relatively complex to use.

7. REMARKS

We have presented the design, performance evaluation, and a prototype implementation of a fault-tolerant model for servers of the RHODOS distributed operating system. The model uses two servers, called twins, to provide a particular service. Apart from accepting normal requests from clients, a twin also uses a reserve to keep track of the other twin’s uncompleted incoming requests. If a twin dies, the other twin will invoke a new twin. The clients will be transparent of any twin failures.

The results from simulation show that our model works very well in various situations. When the twin failure probability or the feedback probability is low, or when the traffic intensity of the original system is not too high, our model does not impose too much overhead on the system performance.

Our model has a wide range of applications in which continuous availability of services are required in the case of some components failures. Apart from applications in the kernel/service pool-based distributed operating systems, other application areas could be, for example, supervisory and telecontrol systems, switching systems, process control and data processing.

ACKNOWLEDGMENTS

The authors would like to thank the anonymous referees for their valuable comments.

REFERENCES


Chang, R. N., and Ravishankar, C. V. A Service Acquisi-


Chapter 43

An analysis of the Web-based client-server computing models

Wanlei Zhou, School of Computing/Maths, Deakin University, Geelong, VIC 3217, Australia.
wanlei@deakin.edu.au

Andrzej Gosciniski, School of Computing/Maths, Deakin University, Geelong, VIC 3217, Australia.
ang@deakin.edu.au

Abstract

This paper discusses models of Web-based client-server computing systems. We classify Web-based client-server computing systems into four groups and describe them using analytic models. We carry out a performance study of each model based on the changes of various parameters related to applications and components of the Web-based computing systems in order to determine basic properties of these four systems.

Keywords: Web-based client-server computing, performance evaluation, queuing theory.

1 Introduction

The Internet and WWW have influenced distributed computing by the global coverage of the network, Web servers distribution and availability, and architecture of executing programs. The question is what model should be used to develop application and system software of Web-based distributed computing systems. We claim here that a natural model of Web-based distributed is the client-server model [Gosciniski and Zhou]. Following this, the current image of Web-based distributed computing can be called the Web-based client-server computing.

However, it is not good enough to use the simple client-server model to describe various components and their activities of a Web-based client-server computing system. The Internet, and in particular Web browsers and further developments in Java programming, have expanded the client-server computing and systems. This is manifested by different forms of co-operation between remote computers. The issue is which form is best. For this purpose, this paper tries to classify various Web-based client-server computing systems into four types, build appropriate models for each type, and carry out performance analysis of these models under various conditions related to applications and systems.

2 Web-based client-server computing models

We have categorised the Web-based client-server computing systems into four types: the proxy computing model, the code shipping model, the remote computing model and the agent-based computing model.

The proxy computing (PC) model is typically used in Web-based scientific computing. According to this model the client sends data and program to the server over the Web and requests the server to perform the computing. The server receives the request, performs the computing using the program and data supplied by the client and returns the result back to the client. Typically, the server is a powerful high-performance computer or it has some special system programs (such as special mathematical and engineering libraries) that are necessary for the computing. The client is mainly used for interfacing with the user. Figure 1(a) depicts this model.

The code shipping (CS) model is a popular Web-based client-server computing model. A typical example is the downloading and then execution of Java applets on Web browsers, such as Netscape Communicator and Internet Explorer. According to this model, the client makes a request to the server, the server then ships the program (e.g., the Java applets) over the Web to the client and the client executes the program (possibly) using some local data. The server acts as the repository of programs and clients perform the computation and interface with the user. Figure 1(b) illustrates this model.
The remote computing (RC) model is typically used in Web-based scientific computing and database applications [Sandewall]. According to this model, the client sends data over the Web to the server and the server performs the computing using programs residing in the server. After the completion of the computation, the server sends the result back to the client. Typically the server is a high-performance computing server equipped with the necessary computing programs and/or databases. The client is responsible for interfacing with the user. The NetSolve system [Casanova and Dongarra] uses this model. Figure 1(c) depicts this model.

The agent-based computing (AC) model is a three-tier model. According to this model, the client sends either data or data and programs over the Web to the agent. The agent then processes the data using its own programs or using the received programs. After the completion of the processing, the agent will either send the result back to the client if the result is complete, or send the data/program/midium result to the server for further processing. In the latter case, the server will perform the job and return the result back to the client directly or via the agent. Many commercial systems are based on this model [Chang and Scott, Ciancarini et al]. Figure 1(d) shows this model.

![Diagram](image)

**Figure 1. Models for Web-based client-server computing**

The question arises from the above classification is which Web-based client-server computing model is best for an application? i.e., the application that the user is to deploy using the Web. The answer to this question actually depends on many parameters. We are interested in two types of parameters. (1) Parameters related to applications: Data size $D$, Program size $P_i$; Result size $R_i$; Expected execution time of program $P_i$; Request size $R_q$; Expected execution time of request $R_q$. (2) Parameters related to various components of the Web-based computing systems: Mean time service rate of the server $\mu_S$; Mean time service rate of the communication system $\mu_CS$; Mean time service rate of the agent $\mu_A$; Mean time service rate of the client $\mu_C$.

For the PC model, we are interested in $D$, $P_i$, $R_i$, $\mu_S$, and $\mu_CS$. $D$, $P_i$, and $R$ represent the load to the communication system. $P_i$ represents the load to the server, and $\mu_S$ and $\mu_CS$ represent the service capabilities of the server and the communication system, respectively. For the CS model, the parameters are $R_q$, $R_i$, $P_a$, $P_n$, $\mu_S$, $\mu_CS$, and $\mu_C$. $R_q$ and $P_q$ represent the load to the server. $P_i$ represents the load to the client, and $\mu_S$, $\mu_CS$ and $\mu_C$ represent the service capabilities of the server, the communication system and the client, respectively. For the RC model, the parameters are $D$, $P_i$, $R_i$, $\mu_S$, and $\mu_CS$. $D$ and $R$ represent the load to the communication system. $P_i$ represents the load to the server, and $\mu_S$ and $\mu_CS$ represent the service capabilities of the server and the communication system, respectively. For the AC model, the parameters are $D$, $P_a$, $R_i$, $\mu_S$, $\mu_A$, $\mu_CS$, and $p$. $D$, $P_n$, and $R$ represent the load to the communication system. $P_i$ represents the load to the agent and the server, $\mu_S$, $\mu_A$, and $\mu_CS$ represent the service capabilities of the server, the agent and the communication system, respectively. $p$ is the probability that a $\text{Data}$ or a $\text{Data, Program}$ message can be serviced solely by the agent. We are also interested in the changes of the response time of each model when the parameters of applications and systems change.
3 The analytic models

By analysing Figure 1(a) and Figure 1(c), we can see that the PC and RC models have many common features. Firstly, both of them require message \(<\text{Data, Program}\) or \(<\text{Data}\) transmission from the client to the server and the message \(<\text{Result}\) transmission from the server to the client. The two message transmissions must be handled by the communication system. Secondly, both of them also require the server to carry out computing using the supplied data. The only difference is that the PC model requires the server to use the program supplied by the client while the RC model uses the program stored in the server. Therefore, we can combine the two models into one analytic model, depicted in Figure 2(a).

![Diagram](image)

Figure 2: The analytic model for the PC and RC models.

In this model, the \(\lambda_{C-CS}\) and \(\lambda_{S-CS}\) represent the load of the communication system. In the case of the PC model, the \(\lambda_{C-CS}\) represents the size of the data and the programs \((D \text{ and } P_s)\). For the RC model, the \(\lambda_{C-CS}\) is the size of the data \((D)\). For both models, the \(\lambda_{S-CS}\) represents the size of the result \((R)\). The \(\lambda_{CS-S}\) represents the load to the server \((P_s)\).

Similarly, an analysis of Figure 1(b) can lead us to the analytic model of the CS model as shown in Figure 2(b). In this model, the \(\lambda_{C-CS}\) and \(\lambda_{S-CS}\) represent the load of the communication system. The \(\lambda_{C-CS}\) represents the sizes of the requests \((R)\) and the \(\lambda_{S-CS}\) represents the sizes of the programs shipped to the client \((P_s)\). The \(\lambda_{CS-S}\) represents the load to the server \((P)\). The \(\lambda_{CS-C}\) represents load to the client \((P)\).

An analysis of Figure 1(d) results in the analytic model of the AC model, shown in Figure 2(c). In this model, the \(\lambda_{C-CS}\), \(\lambda_{A-CS}\) and \(\lambda_{S-CS}\) represent the load to the communication system. The \(\lambda_{C-CS}\) represents the sizes of the data \((D)\) or the data and the programs \((D \text{ and } P_s)\). The \(\lambda_{A-CS}\) and \(\lambda_{S-CS}\) represent the sizes of the results \((R)\). The \(\lambda_{CS-A}\) and \(\lambda_{CS-S}\) represent the load to the agent and the server \((P)\).

Solving the above analytic models is a very difficult task since the basic assumption, the flow balance assumption [MacDougall] for solving queuing networks is not true in these models. Thus, a further simplification is needed to solve these analytic models. We assume that each queue in the analytic models is a M/M/1 queue and all the queues are independent of each other (that is, the flow imbalance feature does not affect our analysis). With this assumption, the average response time of each model can be calculated using the Jackson Theorem [Kant]. We omit the detailed analysis here.

4 Result analysis

Figure 3(a) shows the change of the average response time for the PC and RC models when the load \(\lambda_{C-CS}\) or...
\( \lambda_{CS-s} \) changes, for the service rates \( \mu_{CS}=20 \) and \( \mu_s=25 \). It can be seen that when the load to communication system increases (the curve marked by \( L_{cs} \)), the response time increases slowly until it approaches the limit of the communication system (\( \lambda_{CS-s}=20 \)), in that case the response time increases dramatically. We omit the case for \( \lambda_{s-CS} \) since the calculation result is similar to the case of \( \lambda_{CS-s} \). When the load to the server increases (the curve marked by \( L_{s} \)), the response time increases slowly until it approaches the limit of the server (\( \lambda_{CS-s}=25 \)), in that case the response time increases sharply. Thus we conclude that the PC and RC models show a reasonably good performance if we have light load to communication system (i.e., the sizes of the <Data, Program>, <Data>, and <Result> are small) and to server (i.e., the program's expected execution time is short).

![Graphs showing response time changes](image)

**Figure 3. The average response time when load or service rate changes.**

Figure 3(b) shows the changes of the average response time for the PC and RC models when the service rate changes, for \( \lambda_{CS-s}=10 \), \( \lambda_{CS-s}=10 \) and \( \lambda_{s-CS}=10 \). It can be seen that when the communication system's service rate is low (the curve marked by \( M_s \)), the response time is very long. However, the response time decreases dramatically and then stabilises when \( \mu_{CS} \) increases. Similarly, when the service rate of the server is low (the curve marked by \( M_s \)), the response time is very long. As \( \mu_s \) increases the response time decreases sharply.
and then becomes stabilised. We can then conclude that the PC and RC models have reasonably good performance if the communication system and the server are fast.

Figure 3(c) shows the calculation results for the CS model, for \( \mu_{CS} = 20, \mu_S = 25 \) and \( \mu_c = 22 \) when the load (\( \lambda_{CS}, \lambda_{CS,S}, \lambda_{CS,C} \)) changes. We omit the case for \( \lambda_{CS} \) since it is the same as the case of \( \lambda_{CSE} \). Figure 3(d) shows the changes of the response time when the service rate (\( \mu_{CS}, \mu_S \) or \( \mu_c \)) changes, for \( \lambda_{CS} = 10, \lambda_{CS,S} = 10, \lambda_{CS,C} = 10 \) and \( \lambda_{CSE} = 10 \).

The calculation results for the AC model, for \( \mu_{CS} = 20, \mu_S = 25 \) and \( \mu_A = 22 \) during the calculation of the average response time when the load (\( \lambda_{CS}, \lambda_{CS,A}, \lambda_{A,CS}, \lambda_{CS} \)) changes, are shown in Figure 3(e). We also omitted the case for \( \lambda_{CS} \) as before. Figure 3(f) shows the changes of the response time when the service rate (\( \mu_{CS}, \mu_S \) or \( \mu_A \)) changes, for \( \lambda_{CS} = 10, \lambda_{CS,A} = 10, \lambda_{A,CS} = 10, \lambda_{CS,S} = 10, \lambda_{CS,C} = 10 \). In both cases, \( p = 0.5 \).

From Figure 3(e) and Figure 3(f) we can see that for the CS and AC models the response time increases as the load increases. The increment of the response time becomes dramatic when the individual load approaches its corresponding service rate limit. From Figure 3(d) and Figure 3(f) we can see that the response time decreases dramatically and then is stabilised as the service rates increase.

We can conclude from Figures 3(a)-(f) that the all the models have good performance if the loads are far lower than their corresponding service rates.

![Graphs showing response time and service rate changes](image)

**Figure 4. Comparison of models.**

**Figure 5. The average service time when \( p \) changes.**

Figure 4 compares the performance of our models. It can be seen that the four models do not have significant difference in performance, although the PC and RC models show slightly better performance since they do not require many stages of computation and communication. It is interesting that the CS and AC models have
very similar performance though the AC model is considerably more complex than the CS model. The reason
here is that we have assumed $p=0.5$, i.e., 50% of the requests from clients will be served by the agent only.
This can off-set the delay incurred by the other 50% of the requests served by both the agent and the server.

It is then interesting to see the influence of $p$ on the AC model. Figure 5 shows this effect. It can be seen that
when $p$ is large, the response time is short if the load is not very heavy (e.g. $p=0.9$). The response time
increases as $p$ decreases. This is because that when $p$ is large, most of the request will be treated by the agent
only. As $p$ increases, the number of requests processed by both agent and server increases, therefore the
response time increases. It is also interesting to see that as the load ($\lambda_{AC}$) increases, the curve marked by
$p=0.9$ overtakes the curve marked by $p=0.5$. The reason is that when too many requests are passing through
the communication system (i.e., $\lambda_{AC}$ is large and $p$ is large) at the same time, it becomes a bottleneck. The
case of $p=0.5$ evenly distributes the traffic between the communication systems and agents/agents.

5 Conclusions

In this paper we have classified Web-based client-server computing systems into four computing models, i.e.,
the proxy computing model, the code shipping model, the remote computing model, and the agent-based
computing model. We also have developed analytic models for these Web-based client-server computing
models and have evaluated their performance. From the analysis of the initial results, we can reach the
following conclusions: (1) All the models can offer good performance when the loads to various service
centres (i.e., the communication system, the server, the agent, and the client) are low and these service
centres have high service rates. (2) Since information can flow through various stages of the models, each
stage has a potential to become a bottleneck of the whole system. The more stages a model has, the higher the
bottleneck potential it has. Therefore, the PC and RC models have the least probability to form a bottleneck,
the CS model has a moderate chance to get a bottleneck, whereas the AC model has the highest chance to
form a bottleneck. (3) The AC model has a reasonably good performance compared to other simpler models.
Nowadays, more and more Web-based applications have shifted to the AC model [Chang and Scott, Duan,
Ciancarini et al]. According to our analysis, it is possible to achieve good performance using this three-tier
(or multi-tier) model as long as the developer is aware of the following design issues: (a) to keep lower traffic
loads, (b) to achieve higher service rates, (c) to develop powerful agents that can meet at least half of the
client request, and (d) to avoid bottlenecks.

References
Using the NetSolve System", Proceedings of the 3rd IEEE International Conference on Algorithms and
ordinate Distributed Applications on the Web", The Fifth International World Wide Web Conference, May 6-
[Duan] N. N. Duan, "Distributed Database Access in a Corporate Environment Using Java", The Fifth
[Goscinski and Zhou] A. Goscinski and W. Zhou, Client-Server Model, The Encyclopedia of Electrical and
[Sandewall] E. Sandewall, "Towards a World-Wide Data Base", The Fifth International World Wide Web
Managing Replicated Remote Procedure Call Transactions

WANLEI ZHOU AND ANDRZEJ GOSCINSKI

School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

This paper addresses the problem of building reliable computing programs over remote procedure call (RPC) systems by using replication and transaction techniques. We first establish the computational model: the RPC transactions. Based on this RPC transaction model, we present the design of our system for managing RPC transactions in the replicated-server environment. Finally, we present some results of a correctness study on the system and two examples of the system.

Received 17 April, 1997; revised 30 August, 1999

1. INTRODUCTION

Two approaches are commonly used in supporting fault-tolerant computing. The first approach provides programming languages that are targeted at developing fault-tolerant systems. Typical examples of this approach are the Ada 95 [1], the fault-tolerant concurrent C [2], and the fault-tolerant version of the SR language [3]. The second approach provides a fault-tolerant computing toolkit or a model that can be used together with general programming languages and standard operating systems. Typical examples of this approach are the ISIS toolkit [4], the ARGUS system [5], the location-based replication paradigm [6], the fulfillment transactions approach [7], and the RPC transaction management system [8]. This paper follows the second approach.

Remote procedure call (RPC) is perhaps the most popular model used in today’s distributed software development and has become a de facto standard for distributed computing. The use of RPC facilitates the building of distributed programs and removes concerns for the communication mechanisms from the programs that use remote procedures. Only fundamental difficulties of building distributed systems such as synchronisation and independent failure of components are left in RPC programming.

Many leading computer companies have agreed on a vendor-neutral distributed computing environment (DCE) architecture proposed by the Open Software Foundation [9]. This architecture is designed under the client/server model, and requires the interactions between its components to follow the RPC paradigm. Although the DCE architecture helps reduce the heterogeneity of server-access protocols and provides a limited fault-tolerant support in the service level, one important issue is still outstanding: the support for fault-tolerant computing from the RPC level.

Since fault tolerance is not provided in the RPC level, system services and user applications running on DCE have to employ their own mechanisms for dealing with reliability and availability of the system. This limitation has resulted in a number of problems such as (1) adding another dimension of difficulties (dealing with fault tolerance) in software development; (2) repeated development of fault-tolerant mechanisms in services and applications; and (3) less efficient fault-tolerant mechanisms since they are running on higher protocol levels.

For example, the Directory Service of the DCE uses a primary copy and a number of read-only copies to provide a distributed and replicated repository for information on various resources of a distributed system. This mechanism has the inconsistency and reconfiguration problems in the case of failures, as described in [10] and [11]. Reference [11] proposes an extension of the DCE Directory Service to provide a better fault-tolerant service. However, it can only solve the fault-tolerance problem on one service.

It has been suggested that the use of replication and transaction techniques can provide an environment for developing reliable programs [12]. Replication is the key to providing high availability, fault tolerance, and enhanced performance in a distributed system. However, although considerable research efforts have been directed towards the design of replication-control protocols, replication is still viewed as a ‘necessary evil’ [6]. Reference [13] gives a comprehensive overview of replication techniques and annotated bibliographies of selected literature on replication techniques and example systems.

Transaction management is a well-established concept in database system research. A transaction is defined as a sequence of operations over an object system (a system with an associated collection of objects, where an object can be a database file, an entry of a database file, or can model a real-world entity such as printers or actuators of a control system), and all operations must be performed in such a way that either all of them execute or none of them do [12]. Transactions are used to provide reliable computing systems and a mechanism that simplifies the understanding and reasoning about programs.
not executed, while with other error conditions it is not clear if the procedure was executed or not. For example, on receiving an 'object not free' or a 'server error', we are sure that the RPC was not performed. But on receiving a 'communication error', the client will not be able to tell if the call was performed or not, because we do not know whether the error happened before or after the calling request arrived at the destination host.

We define the effects of an RPC as the processing of one data item of the object system. Hence we can abstract an RPC as a mapping of the following type:

\[ c : P \times O \rightarrow \{OK, FL, US\}, \]

where \( O \) is the union of all objects managed by all services of the system. The values of the target set have the following meaning:

**OK**: This means that no failure occurred during the RPC's execution. By \( c(p, d) = OK \), where \( p \in P \) is a remote procedure and \( d \in O \) is the data object processed by the call, we mean that the RPC call was successful (i.e. the remote procedure performed the job).

**FL**: This means accessing failure. By \( c(p, d) = FL \) we mean that the destination server (the server that exports the remote procedure \( p \)) could not perform the job because, for example, the arguments between the client and server do not match, the versions are different, or the object managed by the server is not free. This means that the RPC has not executed.

**US**: This means unknown state. By \( c(p, d) = US \) we mean that the client cannot tell if the RPC has or has not been executed because, for example, the destination host (the host that the server exporting procedure \( p \) resides on) is down, or the server is down, or the links between the client and server are down (that is, the client host and the server host belong to two partitioned sub-networks). In that case the RPC request may be lost, or the return message may be lost. So we do not know if the remote procedure has been executed or not.

Without loss of generality we assume that all RPCs are update-oriented operations. That is, if \( c(p, d) \) is successful it transforms the data object \( d \) from the existing state to a new state.

2.3. RPC transaction model

We define a (parallel) RPC transaction as \( T = \{c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k)\} \), where \( c_i(p_i, d_i) \) is an RPC and \( p_i \in P, d_i \in O \). The semantics of an RPC transaction is that after issuing the transaction, all \( c_i \) of \( T \) will be executed if no error occurs (commit or OK), or if any one of them fails, all executed RPCs will be rolled back (abort or FL). An RPC \( c_i(p_i, d_i) \in T \) returns OK if and only if all replicas of \( d_i \) have successfully performed the procedure \( p_i \). Similarly, \( c_i(p_i, d_i) \) returns FL if and only if some of its replicas of \( d_i \) failed to execute the procedure \( p_i \) on \( d_i \).

In addition to the two normal states (commit and abort), we define a parallel commit (PC) state. The real meaning behind the PC state is that a replica is not accessible (e.g. the replica is down or the network is partitioned), but there are other replicas that can provide the same service. So the transaction has only performed RPCs on all replicas that are alive (can be reached now). For those replicas that are not accessible now, the transaction effect will be resolved when these replicas re-join the service (e.g. the failed replica recovers or the network is re-united). During the conflict-resolution phase, all the transactions that returned PC will be checked. If the work of such a transaction does not conflict with any other transactions (e.g. it does not use any common data objects with other transactions), then it is considered to be safe and will be committed. If a conflict is detected between two transactions, then one of them is chosen as a victim (unsafe) and will be aborted. The other will then be safe and committed. This is done asynchronously.

Most existing RPC systems are synchronous in nature, and hence fail to exploit fully the parallelism of distributed applications. Our RPC transaction model is synchronous when there is no failure and is asynchronous otherwise. The model can achieve high parallelism while retaining the simplicity of the RPC abstraction.

The execution of all \( c_i \) in \( T \) is in parallel. Some parallel primitives can be built for parallel execution of remote procedures [18]. Sequentially executed transactions can be easily established from the parallel model. However, if operations within a transaction are to be executed sequentially, the serialisability [19] must be considered. In this paper we only consider parallel RPC transactions. Figure 1 indicates the semantics of an RPC transaction.

Parallel transactions exist in many applications. For example, in a scheduling application a scheduler is responsible to schedule \( k \) \( (k > 0) \) tasks concurrently and the schedule is considered successful if and only if all the tasks are successfully scheduled. Schedulers for meetings, timetables, and even flight reservations (multi-hops) can be modelled as scheduling applications.

If we do not need to distinguish individual RPCs within an RPC transaction \( T \), we then write \( T = \{c_i(p_i, d_i)\} \) and use \( |T| \) to represent the number of parallel RPCs within \( T \).
There have been some efforts to combine two of the three techniques—RPC, replication, and transaction—together to achieve reliable computing. However, none of the existing systems/proposals are completely satisfactory.

The ISIS toolkit [4] is a distributed programming environment, including a synchronous RPC system, based on virtually synchronous process groups and group communication. ISIS combines RPC and replication techniques to achieve the goal of developing reliable programs. A special process group, called a fault-tolerant process group, is established when a group of processes (replicated servers and clients) is cooperating to perform a distributed computation. Processes in this group can monitor one another and can then take actions based on failures, recoveries or changes in the status of group members. A collection of reliable multicast protocols is used in ISIS to provide failure atomicity and message ordering. The drawback of developing fault-tolerant programs using ISIS is the performance penalties incurred by using process groups.

The location-based paradigm for replication proposed by Triantafillou and Taylor [6] addresses the problem of combining reliability with performance issues. It uses replication and transaction techniques to develop reliable programs. The proposal provides reliability similar to quorum-based replication protocols but with transaction delay similar to a one-copy system. However, the proposal cannot deal with network partitions properly. Another paper by the same authors [14] addressed the issue of achieving high availability in a partitioned distributed system through the use of transaction and replication techniques.

The RPC transaction management system proposed by Zhou and Molinar [8] uses RPC and transaction techniques to develop reliable programs. RPCs are grouped into transactions that are guaranteed to be atomic. However, the proposed system does not work in a replicated environment.

The Coda file system [15] supports the replication of file volumes, managing the resulting multiple-update problem even in the presence of server failures. The replication of file volumes produces a fault-tolerant service. The most successful feature of the Coda system is its support for disconnected operations. However, the reintegration of files after a disconnection involves manual intervention in the case of conflicting updates.

The fulfillment transactions approach adopted by the Totem group [7] addresses the issue of reliable transaction management in a replicated environment, through the use of the Totem multiset group communication system [16]. Transactions normally commit even in the presence of system failures (e.g. during a network partition), and corresponding fulfillment transactions are generated to record the details of these affected transactions. These fulfillment transactions will then be processed by the system when the failure is recovered in order to bring all replicas to a consistent state. However, fulfillment transactions are application-specific and in some cases, human intervention is needed to reconcile the inconsistencies.

The purpose of this paper is to combine replication, transaction management and RPC together to form a reliable and efficient distributed computing environment. Fault-tolerant programs developed in our environment should be able to tolerate single failures such as a server failure, a site failure or even a network partition without involving manual intervention. These programs should also be efficient and should not incur too much overhead compared with a non-replicated system when there are no component failures.

The remainder of the paper is organized as follows. Section 2 introduces the replica, the RPC model, the transaction model and the failure semantics. Section 3 presents the model and the algorithm for transaction management. Section 4 describes the replica management. Section 5 discusses the conflict-resolution algorithms. Section 6 presents some results of a correctness study of the proposed system. Section 7 describes an illustration example and an application example of the system. Section 8 concludes the paper.

2. SYSTEM MODELS

2.1. Replicas

A distributed system with replicated servers consists of many sites interconnected by a communication network. A service is provided by a group of replicated servers (called replicas) executing on some sites. These replicas manage some common data objects that can be shared by many clients. For simplicity, we assume that each replica knows the location of other replicas that store the same data objects. This assumption can be loosened if a replication directory service is used.

When requesting a service, a client specifies the service through, say, an attributed name [17]. The exact location of the service and the server that provides such a service will be determined by the system.

We model a service as a set of replicas \( S = \{ S_1, S_2, \ldots, S_i, \ldots, S_n \} \), where \( i = 1, 2, \ldots, n \), are called the sequence numbers of these replicas. Each replica \( S_i \) manages a set of data objects \( O_i = \{ d_1^i, d_2^i, \ldots, d_n^i \} \). The consistency constraint requires that for \( i, j = 1, 2, \ldots, n \) and for each service \( S_i, O_i \equiv O_j \).

2.2. The RPC model

Each replica in our system provides a number of remote procedures that can be called by clients for processing the data objects managed by the replica. We use \( P \) to denote the set of all remote procedures provided by all replicas of the system:

\[ P = \{ p \mid p \text{ is a remote procedure of the system} \} \]

Our RPC model has the exactly-once call semantics in the absence of failures and the at-most-once call semantics otherwise [17]. In particular, after we make an RPC the call may return successfully or fail. There may be several reasons for the failure of a call such as ‘object not free’, ‘server error’ or ‘communication error’. With some error conditions it is clear to the client that the procedure was...
2.4. Failures

There are four classes of failures in a replicated-server environment executing RPC transactions: transaction abort, replica-is-down, site-is-down, and network partition.

An RPC transaction \( T \) aborts if any of its RPC returns an FL (e.g., the requested data object is not free, or if all replicas of a service that the RPC accesses are down), or \( T \) was in a PC state and then the conflict-resolution process identifies that \( T \) is unsafe. The usual treatment for a transaction failure is to retry it after a random period of delay. The random delay is necessary to avoid oscillation when two or more transactions need to use the same data object at the same time.

A replica being down means that the replica is not accessible. In this case, other replicas of the same service should still provide the service to clients (an RPC returns FL if all replicas of a service that it accesses are down). However, when the failed replica recovers and rejoins the service, it may have stale information because some data objects may have been updated when the replica was down. To solve this problem, we set all transactions on the five replicas of the service to the PC state. These partially committed transactions will be made to commit when the failed replica rejoins the service and starts the recovery process. At the same time, the failed replica will update all of its stale information.

A site being down means that the replica(s) that runs (run) on it is (are) down. We assume that each site has only one replica running on it. That reduces the problem into a replica-is-down failure.

A network partition failure means that replicas of a service may belong to two disconnected partitions. In this case, the two sets of replicas should still provide the service to clients. However, when the two partitions are reunited, the two sets of replicas may have performed some conflicting operations. To solve this problem, we also set all transactions performed during the network partition on these two parts of replicas to PC states. These partially committed transactions will be resolved when the two parts are reunited.

We use \textit{system failures} to denote the latter three failures and we assume a single failure for system failures. That is, at any time only one of the replica-is-down failure, or the site-is-down failure, or the network-partition failure occurs.

As discussed above, we can actually classify them into two types of failures: the \textit{replica failure} and the \textit{partition}. A replica failure means that a replica is down or the site that the replica is running on is down. A partition means the network is partitioned into two disconnected sets and both parts have some replicas running on them. In both cases, we can divide the replicas into two parts. For simplicity, we only assume crash failures in this paper. Byzantine failures can also be dealt with when the number of replicas of a service is more than three and a majority voting scheme is used.

3. RPC TRANSACTION MANAGEMENT

3.1. The RPC transaction processing model

We define a \textit{primary replica} for a data object \( d \) as a replica that is the best (e.g., the nearest site to the RPC transaction manager (described below) that accepts the client request, although the measure is left for individual applications) in performing an RPC \( c(p, d) \). Any replica can be chosen as the primary replica for a particular RPC. The management of primary replica information is itself a difficult issue. We will concentrate on the transaction management and therefore will ignore this problem here.

Three system components are involved in processing a transaction submitted by a client.

(i) An \textit{RPC transaction manager} (RTM) accepts a transaction \( T(c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k)) \) from the client. The RTM sends each RPC \( c_i(p_i, d_i) \in T \) to a primary replica of \( d_i \) and asks the primary replica to check if the RPC can be performed or not. We denote this operation as \( a(c_i(p_i, d_i)) \). The RTM then acts as a coordinator for managing the atomicity of \( T \) through the help of the primary replicas. Section 3.2 describes the RTM algorithm.

(ii) A primary replica accepts, from the RTM, an RPC \( c_i(p_i, d_i) \) and the request to check the executability of the RPC (the \( a(c_i(p_i, d_i)) \) operation). The primary replica sends the RPC to all replicas of data object \( d_i \) and asks all replicas (including itself) to check if they can execute the RPC (e.g. if \( d_i \) is free). We use \( b(c_i(p_i, d_i)) \) to represent this operation. The primary replica acts as a coordinator for managing the RPC \( c_i(p_i, d_i) \) to be performed on all replicas (including itself). Section 4.1 describes the coordinating algorithm.

(iii) Each replica of the data object \( d_i \) accepts, from the primary replica of \( d_i \), the RPC \( c_i(p_i, d_i) \) and the request to check the executability of the RPC (the \( b(c_i(p_i, d_i)) \) operation). The replica then cooperates with the primary replica by returning the executability check and performing the RPC when requested. Section 4.2 describes the cooperating algorithm.

The request for checking the executability of an RPC is essentially a request for a lock on the data object to be updated by the RPC in most cases. However, in some cases other factors can also affect the executability of an RPC.

The actual effect of an \( a(c_i(p_i, d_i)) \) call depends on the associated \( b(c_i(p_i, d_i)) \) calls. That is, if all \( b(c_i(p_i, d_i)) \) calls return OK, \( a(c_i(p_i, d_i)) \) returns OK. If any \( b(c_i(p_i, d_i)) \) call returns FL, \( a(c_i(p_i, d_i)) \) returns FL. If no \( b(c_i(p_i, d_i)) \) call returns FL and there are PCs returned by \( b(c_i(p_i, d_i)) \) calls, or no FL return but there are US or OK returns (the primary replica must return OK or PC in this case), then \( a(c_i(p_i, d_i)) \) returns PC. An \( a(c_i(p_i, d_i)) \) may also return US if the primary replica for \( d_i \) is not accessible.

A client simply submits a transaction request to the RTM and waits for the transaction result and the resulting state. The resulting state returned at this stage can be one of OK, FL or PC. As soon as the client receives a resulting state of OK or FL, it can continue with its next operation. This means that other operations, such as operations required for consistency or for continuous committing, will be dealt with by the system without the participation of the client.
FIGURE 2. RPC transaction processing.

However, if no primary replica returns an FL but some primary replicas return US (i.e. these replicas are not accessible because of a site failure or a network partitioning), then the RTM will retry to find new primary replicas for those primary replicas that returned US. This process repeats until either there are no more available replicas that can act as primary replicas for some particular data objects, or the newly found primary replicas return OK/PC/FL.

If the transaction returns a PC state, it should enable an asynchronous receive procedure (similar to an exception handler) to process the following asynchronous message returned for the transaction. This message will come when the partially committed transaction is resolved. In this case, either a commit or an abort message will be returned. The asynchronous receive procedure should then process this accordingly. Figure 2 depicts the model for RPC transaction processing.

The number of RTMs in a system depends on the requirements of specific applications. Our examples (Section 7) show two different configurations. The first example (Section 7.1) uses one RTM for the whole system, whereas the second example (Section 7.2) uses three RTMs. A number of strategies, such as the twin-server model [20], can be used to improve the reliability of the RTM.

3.2. The transaction manager

The transaction manager has three basic functions to perform when it receives a transaction T.

(i) It submits operations to their primary replicas. The manager is responsible for locating primary replicas for each RPC and then sending these RPCs to their corresponding primary replicas.

(ii) It manages the atomicity of the transaction T. It uses the 2-phase-commit (2PC) protocol to ensure that if all RPCs of T to the primary replicas return OK, T asks all primary replicas to commit and returns OK. If any of these RPCs return FL, it asks all primary replicas to abort and returns FL. In this case, any uncommitted update will eventually be rolled back. If no operation returns FL, but some operations return US, then the manager will find new primary replicas for those that returned US and try again. If all operations return OK or PC (and there is at least one PC return), then T asks all primary replicas to partially commit and returns PC.

(iii) It sets up an asynchronous receive procedure if a transaction returns a PC. The procedure waits messages from all primary replicas that returned PC. If all of these primary replicas later return OK, then the procedure will upgrade the transaction to a commit state. If any of the primary replicas later returns an FL, then the procedure will downgrade the transaction to an abort.

We define a function locatePrimaryReplica() which takes as its input an RPC and returns the primary replica location of the RPC. If the function returns a NULL, this means that there are no more available replicas that can be a primary replica for the RPC concerned (e.g. all replicas for processing a particular object are cut off from the transaction manager). In this case the transaction has to be aborted. This function also guarantees that if \( c_t(p_1, d_1), c_t(p_1, d_2) \in T \) and \( d_1 = d_2 \), then locatePrimaryReplica(\( c_t(p_1, d_1) \)) = locatePrimaryReplica(\( c_t(p_1, d_2) \)). This means that the two RPCs of the same transaction will use the same primary replica if they access the same data object.

The algorithm manage RpcTransaction() of Listing 1 implements the RTM. The constant MAXRPCS defines the maximum number of RPCs allowed in an RPC transaction.

LISTING 1. Algorithm 1—the RTM algorithm.

```
manage RpcTransactions()
{
    if (Incomplete = TRUE)
    {
        /* receive RPC transactions */
        /* if more transactions come, queue them up until the processing of the current transaction is over */
        receive(client, T = \{ c_t(p_1, d_1) \});
        T' \leftarrow T;
        while (Incomplete) {
            COBEGIN
                primaryAddress[i] = locatePrimaryReplica(d_1);
                \forall c_t(p_1, d_1) \in T';
            COEND;
            if (primaryAddress[i] = NULL, \forall c_t(p_1, d_1) \in T') {
                /* no more available replicas to choose for c_t(p_1, d_1) */
                COBEGIN
                    send(primaryAddress[i], “abort”), \forall c_t(p_1, d_1) \in T;
                COEND;
                tell the client the transaction returns FL;
                InComplete = FALSE;
                break;
            }
            COBEGIN
                ret[i] = a(c_t(p_1, d_1));
                execute on primaryAddress[i], \forall c_t(p_1, d_1) \in T';
            COEND;
        }
    }
    
```
Managing Replicated RPC Transactions

\[ S_{US} = \{ o(c_i(p_i, d_i)) \mid ret_i = US \}; \]
\[ S_{FL} = \{ o(c_i(p_i, d_i)) \mid ret_i = FL \}; \]
\[ S_{PC} = \{ o(c_i(p_i, d_i)) \mid ret_i = PC \}; \]
\[ S_{OK} = \{ o(c_i(p_i, d_i)) \mid ret_i = OK \}; \]

\[
\text{switch} \begin{cases} 
\text{case} \left( S_{OK} \neq \emptyset \land (S_{US} = S_{FL} = S_{PC} = \emptyset) \right): \\
\text{/* all } o(c_i(p_i, d_i)) \text{ executed and returned } OK */ \\
\text{COBEGIN} \\
\text{send(primaryAddress[i], "commits"}, \forall c_i(p_i, d_i) \in T); \\
\text{COEND}; \\
\text{tell the client that the transaction returns } OK; \\
\text{InComplete} = \text{FALSE}; \\
\text{break}; \\
\text{case} \left( S_{FL} \neq \emptyset \right): \\
\text{COBEGIN} \\
\text{send(primaryAddress[i], "abort"}, \forall c_i(p_i, d_i) \in T); \\
\text{COEND}; \\
\text{tell the client the transaction returns } FL; \\
\text{InComplete} = \text{FALSE}; \\
\text{break}; \\
\text{case} \left( S_{FL} = \emptyset \land S_{US} \neq \emptyset \right): \\
\text{/* some of the primary replicas are not accessible */} \\
T' := S_{US}; \\
\text{break}; \\
\text{case} \left( S_{FL} = S_{US} = \emptyset \land S_{PC} \neq \emptyset \right): \\
\text{/* no failure, but there are PC returns */} \\
\text{set asynchronous receive procedure handler; } \\
\text{each primary replica is given the address of the handler} \\
\text{(piggybacked by the following send);} \\
\text{COBEGIN} \\
\text{send(primaryAddress[i], "partial commit"}, \\
\forall c_i(p_i, d_i) \in T); \\
\text{COEND}; \\
\text{tell the client that the transaction returns } PC; \\
\text{InComplete} = \text{FALSE}; \\
\text{break}; \\
\end{cases}
\]

4. REPLICA MANAGEMENT

4.1. The coordinating algorithm for the primary replica

When a primary replica receives an RPC request \( c_i(p_i, d_i) \) from a transaction manager, it uses the coordinating algorithm to maintain the consistency of all replicas in terms of the RPC. This section describes the coordinating algorithm.

In the coordinating algorithm the primary replica uses the 2PC protocol to ensure replication consistency. In the first phase, the primary replica asks all replicas (including itself) to check the executability of the RPC (i.e. the \( b(c_i(p_i, d_i)) \) operation). If all replicas return OK for such an execution, the primary replica returns OK to the transaction manager. If the transaction manager requests commit, then in the second phase the primary replica asks all replicas to commit the RPC execution. If any replica returns FL, then the primary replica returns FL to the transaction manager and asks all replicas to abort the operation in the second phase. If all operations return either OK or PC or US (in this case, the primary replica must return OK or PC and other non-primary replicas may return OK, PC or US), the primary replica returns PC to the transaction manager. The primary replica also records the number of replicas that return OK and PC and the smallest sequence number among these replicas. These two numbers will be used in conflict resolution. If the transaction manager requests partial commit in the second phase, the primary replica asks all replicas to partially commit. If the transaction manager requests an abort, the primary replica then asks all replicas to abort, no matter what was returned by the primary replica in the first phase. The coordinating algorithm is listed in Listing 2. We assume that \( S_i \) is the primary replica.

LISTING 2. Algorithm 2—the coordinating algorithm.

```
primary_replica.process
{
int k, NoOfPCs, smallestPC;
while (TRUE) {
    /* receive RPCs from RTMs */
    /* if more RPCs come, queue up them */
    receive(RTM, a(c(p, d)));
    k = a; /* k is the number of replicas for d */
    Let \( S_j, j \leq k \) be the primary replica;
    COBEGIN
    ret_j = b(c(p, d'));
    where \( d' \) is the copy of d in replica \( S_i, i = 1, 2, \ldots, k \);
    COEND;
    R_{US} = \{ b(c(p, d')) \mid ret_j = US \};
    R_{FL} = \{ b(c(p, d')) \mid ret_j = FL \};
    R_{PC} = \{ b(c(p, d')) \mid ret_j = PC \};
    R_{OK} = \{ b(c(p, d')) \mid ret_j = OK \};
    switch { 
    case \left( R_{OK} \neq \emptyset \land (R_{US} = R_{FL} = R_{PC} = \emptyset) \right): \\
    tell the RTM that the first phase returns OK;
    break;
    case \left( R_{FL} \neq \emptyset \right): \\
    tell the RTM that the first phase returns FL;
    NoOfPCs = |R_{PC}|;
    smallestPC = min(|\{ b(c(p, d')) \} \in R_{OK}|);
    break;
    case \left( R_{FL} = \emptyset \land (R_{US} = R_{FL} = R_{PC} = \emptyset) \right): \\
    tell the RTM that the first phase returns PC;
    NoOfPCs = |R_{PC}| + |R_{OK}|;
    smallestPC = min(|\{ b(c(p, d')) \} \in R_{PC} \cup R_{OK}|);
    break;
    } 
    receive(RTM, command);
    switch { 
    case (command is "commit");
    COBEGIN
    send(S_i, "commit"), i = 1, 2, \ldots, k;
    COEND;
```
break;
case (command is "partial commit"):
COBEGIN
    send(S, "partial commit", NoOfPCs, smallestPC),
    i = 1, 2, ..., k;
COEND;
break;
case (command is "abort"):
COBEGIN
    send(S, "abort"), i = 1, 2, ..., k;
COEND;
break;
}
}

In addition to the coordination work, the primary replica has to do all the work of a non-primary replica. The next section describes the role performed by a non-primary replica.

4.2. The cooperating algorithm for all replicas

When a non-primary replica receives a request from a primary replica, it checks whether the request can be proceeded or not and acts accordingly.

(i) If the request can be performed, the non-primary replica locks the required data object and returns OK. Later, if the primary replica asks to commit the operation, the non-primary replica performs the operation and releases the lock. If the primary replica asks to abort the operation, the non-primary replica releases the lock. If the primary replica asks to partially commit the operation, then the non-primary replica partially commits the operation and records this event.

(ii) If the non-primary replica finds that the operation cannot be executed (e.g. the required data object is not free), it then returns an FL.

(iii) If the non-primary replica finds that the data item is already in a partially committed state, then it returns PC to the primary replica. Later, if the primary replica asks to partially commit the operation, the non-primary replica then partially commits the operation and records the event. If the primary replica asks to abort the operation, then the non-primary replica aborts the operation.

The 2-phase-commit protocol used by the RPC transaction manager guarantees that the replicas will stay in a consistent state if a transaction returns an OK or an FL. However, to guarantee that all replicas do the same thing to the transactions with PC pending, a consensus should be reached among all replicas. This is a classic consensus problem in fault-tolerant computing [21]. We use a need-to-do (NTD) table, which essentially is a checkpointing log, to record the events of partially committed RPCs. Then, during a recovery we let replicas exchange their NTD table entries in order to reach a consensus for actions on the transactions with PC returns. The NTD table of each replica is kept in stable storage [22]. Therefore, information stored in the NTD table will not be affected by system failures. The NTD table structure is listed in Listing 3.

### Listing 3. The need-to-do table.

typedef struct ntd {
    char *rpc; /* name of the partially committed RPC */
    char *data; /* data object name used in this RPC */
    int pc; /* No. of replicas partially committed this RPC */
    int sm; /* smallest sequence # among all PC replicas */
    void *ori; /* before image of the data object */
    void *handler; /* asynchronous receive handler address */
    NTD *pre; /* the previous PC RPC for this data object */
    NTD *next; /* the next PC RPC for this data object */
} NTD;

When the primary replica asks for a partial commit for an RPC, all replicas (including the primary replica) will record this event into their own NTD tables as a new entry. If t ∈ NTD is such an entry, then *rpc contains the name of the RPC, *data contains the data object used in the RPC, *pc stores the number of replicas that have partially committed this RPC, and *sm stores the smallest sequence number among all the replicas that have partially committed the RPC. These two numbers (*pc and *sm) are sent by the primary replica to each replica when it asks for partial commit (see Listing 2). They are used later by the conflict resolution algorithms to determine which partial commit should be upgraded to a commit, and which partial commit should be downgraded to an abort if a conflict occurs.

In order to downgrade a PC to an abort, a before image of the data object is kept in the NTD table. *ori is used to record the address of the before image.

The order of each partially committed RPC over a data object is also very important when a recovering replica carries out these RPCs. We use a pair of pointers to record this order. The *pre stores the previous partially committed RPC and the *next stores the next partially committed RPC for the same data object. The *handler stores the address of the asynchronous receive handler if the replica is the primary replica.

We associate with each data object d a lock d.lock and a partial commit flag d.pc. The actual effect of b(c(p, d)) on d is to check or change the values of d.lock and d.pc. That is, if d.lock = LOCKED (-1), b(c(p, d)) returns FL. If d.lock = d.pc = FREE (0, free and not partially committed), b(c(p, d)) returns OK and sets d.lock = LOCKED. If (d.lock = FREE) ∧ (d.pc = n > 0) (free but partially committed), b(c(p, d)) returns PC and sets d.lock = LOCKED and d.pc = n + 1. If the replica is down or the network is partitioned during the operation (i.e. the replica is unreachable), a US is returned by the RPC system.

All RPCs of a transaction are actually performed in the second phase of the algorithm. During this phase, each replica has to release the lock d.lock (set to FREE) if the c(p, d) has a normal commit or a partial commit. An entry about this partial commit is also inserted into the NTD table. The conflict resolution algorithms are then responsible for
upgrading it to a normal commit or for downgrading it to an abort. The cooperating algorithm is given in Listing 4.

**LISTING 4. Algorithm 3—the cooperating algorithm.**

```c
all.replica.process()
{
    NTD t;
    int myState;
    while (TRUE)
        /* receive RPCs from primary replicas */
        /* if more RPCs come, queue up them */
        receive(PR, b(c(p, d)));
        switch
            case (d.lock = FREE ∧ d.pc = 0):
                d.lock = LOCKED; myState = OK;
                tell the primary replica that the first phase returns OK;
                break;
            case (d.lock = LOCKED):
                myState = FL;
                tell the primary replica that the first phase returns FL;
                break;
            case (d.lock = FREE ∧ d.pc > 0):
                d.lock = LOCKED; d.pc + = 1; myState = PC;
                tell the primary replica that the first phase returns PC;
                break;
        /* US is returned if a replica is unreachable */
        receive(PR, command, NoOfRPCs, smallestPC);
        switch
            case (command is "commit"):
                do (ret = c(p, d)) until ret = OK;
                d.lock = FREE;
                break;
            case (command is "partial commit"):
                d.lock = FREE;
                if myState = OK then d.pc = 1; endif;
                t.rpc = c(p, d); t.pc = NoOfRPCs;
                t.sm = smallestPC; t.data = d;
                t ori = beforeImagered(); t.pre = previous(d); t.ast = 0;
                if the replica is the primary replica of this RPC then
                    t.handler = the asynchronous receive procedure handler
                    for this RPC;
                    endif;
                    do (ret = c(p, d)) until ret = OK;
                    break;
            case (command is "abort"):
                if myState = OK then d.lock = 0;
                else if myState = PC then d.lock = 0; d.pc = - 1;
                endif;
                break;
        }
}
```

5. **CONFLICT RESOLUTION**

An RPC returns US only when the server is down or the server is unreachable (e.g. the network is partitioned). If the server is down, it will eventually be repaired and return to service. In that case, the server missed all updates to its data objects while it is down. Fortunately, we have set these updates in 'partial commit' state and recorded them in NTD tables. Therefore, the conflict resolution algorithms can use this information and make the data objects managed by this failed replica in line with all other replicas. If the network is partitioned into two disconnecting parts, the two parts will eventually be reunited again. In this case, replicas in both partitions may have some partially committed updates. The conflict resolution algorithms are also responsible for making replicas in these two parts consistent.

Since the recovery process is highly critical, we assume that during the recovery process (1) no system failures will happen, and (2) no RPC transactions are allowed. These assumptions can be loosened if more communication and stable storage are used in conflict resolution. For instance, if system failures do happen during the recovery process, then we require that the recovery process logs all its work in a stable storage where the next recovery process can continue based on the stored information. If RPC transactions are allowed to be submitted to a replica during the recovery process, then these transactions will be stored in a stable storage on the replica. The clients will be notified of possible delays for the transactions. When the recovery process completes, these stored transactions will be processed immediately.

When recovering from a replica failure, we assume that there is a process that deletes all entries of the NTD table of the recovering replica, except those entries with t.handler ≠ 0. The reasons for doing this are: (i) those entries with t.handler ≠ 0 are no longer useful—they will be dealt with by other alive replicas; and (ii) those entries with t.handler ≠ 0 have to be dealt with by the recovering replica because it was the primary replica for the particular RPC. This NTD table is sent to all alive replicas by the recovering process.

Two algorithms are needed during the recovery of a system failure. When recovering from a replica failure, the recovering replica has to get a 'reuniting' message (it includes the sequence number of the recovering replica) to other alive replicas. This enables the alive replicas to send outstanding NTD table entries to the recovering replicas by using the send.out.ntd() algorithm.

When recovering from a network partition, replicas of each part of the partition have to send a 'reuniting' message (it includes the sequence numbers of all replicas of the partition) to replicas of the other part. This enables the exchange of NTD tables among replicas in both parts by using the send.out.ntd() algorithm.

Once all replicas have received the NTD table from the other part, the conflict.resolution() algorithm is used by all replicas to resolve possible conflicts and to finalize the outstanding partially committed transactions.
The `send.out.ntd()` algorithm is very simple. It detects the reuniting messages. If such a message comes, it then sends all entries of the NTD table to all replicas included in the reuniting message. Listing 5 describes the algorithm.

**LISTING 5. Algorithm 4—the algorithm for sending out an NTD table.**

```plaintext
send.out.ntd()
{
    while (TRUE) {
        * detect the reuniting messages */
        i* the process suspends until such a message comes */
        reUnitSet = {sequence numbers of reuniting replicas};
        while the NTD table is locked
            wait;
        COBEGIN
            send(Si, NTD), i ∈ reUnitSet;
        COEND;
    }
}
```

The `conflict.resolution()` algorithm is used by all replicas to resolve potential conflicts for a data object. The algorithm receives NTD entries from replicas of the other part. Only the first arriving NTD table is accepted by the algorithm and others are ignored. The algorithm then checks the received NTD entries against its own NTD entries to resolve potential conflicts.

We define a leader of an NTD table as the first partial commit entry for a data object d. That is, if t ∈ T is a leader in an NTD table T, then t.previous = 0. The algorithm uses three functions for processing NTD table entries led by a leader:

(i) The `abort.all(lme, q)` function aborts all RPCs led by the leader q of the NTD table `lme`.
(ii) The `commit.all.own(lme, q)` function commits all RPCs led by the leader q of the NTD table `lme`. The NTD table `lme` is the replica's own NTD table.
(iii) The `commit.all.received(lot, t)` function commits all RPCs led by the leader t of the NTD table `lot`. The NTD table `lot` is the received NTD table.

We first describe these three functions.

Listing 6 is the `abort.all(lme, q)` function. It finds the tail of the entries led by q and rolls back all RPCs from the tail to the leader. If the replica is also the primary replica of the RPC, an FL message is also sent to the asynchronous receive procedure.

Listing 7 is the `commit.all.own(lme, q)` function. It finds the tail of the entries led by q and commits all RPCs from the tail to the leader. If the replica is also the primary replica of the RPC, an OK message is also sent to the asynchronous receive procedure.

Listing 8 is the `commit.all.received(lot, t)` function. It finds the tail of the entries led by t and commits all RPCs from the tail to the leader.

Listing 9 describes the `conflict.resolution()` algorithm.

**LISTING 6. The `abort.all()` function.**

```plaintext
abort.all(lme, q) /* abort all RPCs led by q */
{
    NTD h, t;
    h = q; t = the tail of the NTD entries led by h;
    while (TRUE) {
        d = t.prev; d.pc = 1;
        /* other activities for rolling back an RPC omitted */
        if t.handler ≠ 0 then
            send an FL message to t.handler;
        endif;
        t = t.prev; /* the previous entry */
        if (t.prev = 0) then break; endif;
    }
    delete all entries led by q;
}
```

**LISTING 7. The `commit.all.own()` function.**

```plaintext
commit.all.own(lme, q) /* commit all RPCs led by q, own NTD table */
{
    NTD h, t;
    h = q; t = the tail of the NTD entries led by h;
    while (TRUE) {
        d.pc = 1; t.prev;
        if t.handler ≠ 0 then
            send an OK message to t.handler;
        endif;
        if (t.prev = 0) then break; endif;
    }
    delete all entries led by q;
}
```

**LISTING 8. The `commit.all.received()` function.**

```plaintext
commit.all.received(lot, t) /* commit all RPCs led by t, received NTD table */
{
    NTD h, a;
    h = q; a = the tail of the NTD entries led by h;
    while (TRUE) {
        a = a.prev;
        if (a.prev = 0) then break; endif;
    }
}
```

**LISTING 9. Algorithm 5—the conflict resolution algorithm.**

```plaintext
conflict.resolution()
{
    NTD l, q;
    while (TRUE) {
        /* receive NTD entries from alive/ree-uniting replicas */
        /* the process suspends until such a message comes */
        /* if more such messages come, only accepts the first one */
        /* others ignored. Received NTD entries are stored in lme */
        receive(lme);
        Let lme be the own NTD table of this replica;
        for each leader t of lme do
            if (3 a leader q ∈ lme such that t.data = q.data) then
                /* conflict updates */
```
switch {
    case (t.pc > q.pc) && (t.pc == q.pc && (t.smallestPC > q.smallestPC)):
        /* abort all RPCs led by q in this replica */
        abortAll(t.me, q);
        /* commit all RPCs led by i for other replicas */
        commitAllReceived(tot, i);
        break;
    case (t.pc < q.pc) && (t.pc == q.pc && (t.smallestPC < q.smallestPC)):
        /* commit all RPCs led by q in this replica */
        commitAllOwn(t.me, q);
        break;
    } else commitAllReceived(tot, i); /* no conflict */
endif;
endfor;
for each leader q of t.me do
    commitAllOwn(t.me, q); /* no more conflicts */
endfor;
} delete all entries of t.me (own NTD table);
}

6. CORRECTNESS

In this section we outline the informal analysis of the correctness. We first assume the life-cycle of the system entities (sites, links and replicas) is

work → crash → repair and restart → work.

Without loss of generality, we assume that the maximum down time (including crash, repair and restart time) is finite, and it is denoted as T_d.

**Assertion 1.** If a transaction returns OK, all its RPCs have been executed successfully.

**Proof.** The only way that a transaction returns OK is that in Algorithm 1 we have (S_OK ≠ Ø && (S_US == S_FL == S_PC == Ø)). This means all RPCs to the primary replicas have returned OK. A primary replica returns OK if and only if in Algorithm 2 we again have (R_OK ≠ Ø && (R_US == R_FL == R_PC == Ø)). This means all RPCs to all replicas return OK. A replica returns OK if and only if in Algorithm 3 the data object involved is free and is not partially committed. If the transaction returns OK in the second phase of Algorithm 1 we order all primary replicas to commit the transaction. In this case, all primary replicas will order their replicas to commit in the second phase of Algorithm 2, and therefore all replicas will successfully perform the real RPC in their second phase of Algorithm 3.

**Assertion 2.** If a transaction returns FL, no RPCs of the transaction have been executed.

**Proof.** There are two ways that a transaction could return an FL. The first way that a transaction returns FL is that in Algorithm 1 we have (R_FL ≠ Ø). This means some of the primary replicas returned FL. A primary replica returns FL if and only if in Algorithm 2 some of its replicas returned FL. Furthermore, a replica return FL if and only if in Algorithm 3 it finds that the data object is not free. If the transaction returns FL, in the second phase of Algorithm 1 we order all primary replicas to abort the transaction. In this case, all primary replicas will order their replicas to abort in the second phase of Algorithm 2, and therefore in Algorithm 3 those replicas that returned OK will release their locks to the data object. The replicas that returned PC will set the associated integer of the data object to the free state. The replicas that returned FL will do nothing. In any case, none of the RPCs is executed.

The second way that a transaction returns an FL is that in Algorithm 1 we cannot find any more replicas to be the primary replica for a particular RPC. In this case all primary replicas are ordered to abort the transaction. The rest of the work done by Algorithms 2 and 3 is the same as described above. Therefore, none of the RPCs is executed in any way.

**Assertion 3.** The NTD table will not grow indefinitely and any entry of the table will be deleted eventually.

**Proof.** An NTD table increases if and only if there is a system failure, such as a replica or a site is down, or the network is partitioned into two parts. From our life-cycle assumption for system entities, we know that the maximum time of a system failure is T_d. After that, the replica or the site will rejoin the service and the partitioned network will be reunited. During the recovery, all replicas have to execute Algorithm 5 and the algorithm will delete all NTD table entries at the end of conflict resolution (it is easy to show that Algorithm 5 terminates).

**Assertion 4.** After the conflict resolution, all outstanding PCs will be either committed or aborted.

**Proof.** The conflict resolution is carried out by Algorithm 5 after the exchange of NTD tables between replicas of the two parts (failed or alive replicas, or two parts of the partition). The algorithm checks the replica's NTD table against the received NTD table to see if there are any update conflicts. If no conflict is detected, all PC updates in both NTD tables are committed. If there is a conflict for an update, the algorithm determines the part that has more replicas, or has the smallest sequence number if both parts have the same number of replicas. Then the NTD update from this part is committed, while the NTD update from the other part is aborted. This process continues until all entries of both tables (i.e., all outstanding PCs) are exhausted.

**Assertion 5.** If a transaction returns PC, the transaction will be notified of an OK or an FL return in a finite time.

**Proof.** The only way that a transaction returns PC is that in Algorithm 1 we have (S_FL == S_US == Ø && S_PC ≠ Ø). This means there is no FL nor US return but some primary replicas return PC. A primary replica returns PC if and only if in Algorithm 2 we have (R_FL == Ø && (ret1 == OK || ret2 == PC) && (R_PC ≠ Ø || R_US ≠ Ø)). This means there is no FL return, and the primary replica returns OK or PC (this
guarantees that the primary replica is accessible), and there may be PC or US returns from the replicas. A replica returns PC if and only if in Algorithm 3 the replica finds that the data object is already in a PC state. A replica returns US if it is not accessible. If a transaction returns PC, in the second phase of Algorithm 1 we set up an asynchronous receive procedure and notify each primary replica of the address of this procedure. Then we order all primary replicas to partially commit the RPC. The real effect of a partial commit is in Algorithm 3, where an entry is stored in the NTD table of each involved replica. As we have shown in Assertions 3 and 4, after a finite time all outstanding PCs in all NTD tables will be either committed or aborted. At the same time, these commit/abort messages will be sent by relevant primary replicas to the relevant asynchronous receive procedures (in functions commit.all.own() and abort.all()). These asynchronous receive procedures will then notify the relevant transactions of the final result.

**Assertion 6.** After the recovery of a system failure, all data objects managed by the replicas will be in consistent states.

**Proof.** There are two types of system failure: replica failure and network partition. During the recovery of a replica failure, the recovering replica will receive the NTD entries of an alive replica and all alive replicas will receive an NTD table from the recovering replica. During the recovery of a network partition, replicas of both parts of the partition will exchange their NTD tables. During the conflict resolution, all replicas will perform similar work because they all have the same information (NTD tables of both parts). Entries in these two tables are compared to check conflicts. If no conflicts are found, RPCs in both tables will be committed by all replicas. If any conflict of an RPC occurs, only the RPC from one table will be committed; the other will be aborted. In any case, the result will be consistent.

**Assertion 7.** The proposed system will not impose too much overhead on the system compared with a non-replicated system.

**Proof.** In our system, the transaction returns to the client as soon as it has made the second phase order. The condition for a transaction to make the second phase order is that all primary replicas have returned their feasibility checks (i.e. if the RPCs can be performed). Furthermore, the condition for a primary replica to return the feasibility check is that all replicas of the primary replica have returned the feasibility checks. This means the transaction can return the resulting state to the client without waiting for the RPCs of the transaction to complete. The real RPCs of the transaction are done after these feasibility checks have been returned.

It is anticipated that a feasibility check is faster than a real RPC because the former only checks if the data object is free. In this case, our system has a comparable response time to a non-replicated system, where all real RPCs to all non-replicated servers have to be returned before the transaction can return to the client.

The system response time can even be improved by using the feasibility check results of all primary replicas instead of waiting for the feasibility check result of all replicas of each primary replica. A tentative decision can be made according to these results. An asynchronous procedure is then set up by the transaction manager to process the results returned from all other replicas, and the final transaction result will be decided according to these returns.

### 7. EXAMPLES

In this section we present two examples of the system. The purpose of the first example is to illustrate the algorithms described earlier, and the aim of the second example is to (1) show that our system can be used in practical applications, and (2) experiment with the system in a real implementation to obtain some indications on performance issues.

#### 7.1. An illustrating example

In this example we assume that there are two data objects $d_1$ and $d_2$. Data object $d_1$ is replicated in three replicas $A_1$, $A_2$, and $A_3$, and data object $d_2$ is replicated in $B_1$, $B_2$, and $B_3$. An RPC transaction $T = \{c_1(p_1, d_1), c_2(p_2, d_2)\}$ is to be executed through the help of our RPC transaction manager. Without loss of generality, we also assume that the transaction manager has chosen $A_1$ and $B_1$ as the primary replicas for the execution of the transaction. Figure 3 shows the RPC transaction processing for this example.

Let us discuss the following three typical scenarios. Other situations can be analyzed in the same manner.

**Case 1: No failures.** The RPC transaction manager uses Algorithm 1 to ask primary replica $A_1$ to check the feasibility of $c_1(p_1, d_1)$ and $B_1$ to check the feasibility of $c_2(p_2, d_2)$ (the first phase). The two primary replicas then use Algorithm 2 to check the feasibility of the work. Each replica uses Algorithm 3 to do the job. Since there are no failures, all replicas will return OK and the data objects are locked by individual replicas. In the second phase, the two primary replicas are asked to commit the transaction. In turn, these primary replicas ask their replicas to commit the transaction. In that case, all replicas will commit (the second phase of Algorithm 3) the transaction and release the locks. The transaction is then successfully executed. Figure 4 shows the actions of the RPC transaction manager and the replicas when there are no failures.
FIGURE 4. No failures.

FIGURE 5. The NTD entries during the processing of $T$ and $T'$.

Case 2: $A_2$ is down and then recovers. Let us first look at the situation that $A_2$ is down. In this case the feasibility check by $B_1$ will return an OK, but the feasibility check by $A_1$ will return a PC. All data objects (except $A_2$'s data object) are locked. In the second phase, the RPC transaction manager will decide to partially commit the transaction and each primary replica will be given the address of an asynchronous handler (but only $A_1$ stores the address). All replicas (except $A_2$) will then execute the RPC and release the locks. Also, replicas $A_1$ will set the partial commit number for $d_1 (d_1, pc)$ to 1 and will record the information depicted in Figure 5a into their NTD tables.

$A_3, B_1, B_2$ and $B_3$ will also have a similar entry added into their NTD tables.

When $A_2$ recovers and is reunited into the system, $A_2$'s recovery process will clean up its NTD table and send a reuniting message and the empty NTD table to all other replicas of $d_1$ (i.e. $A_1$ and $A_3$). Upon receiving the reuniting message, $A_1$ and $A_3$ will use Algorithm 4 to send their NTD table entries to $A_2$ (the first arrived NTD table will be accepted, others discarded). Then, all three replicas will use Algorithm 5 to resolve conflicts based on the same view of the NTD tables. In that case, $A_1$ and $A_3$ will commit the RPC that returned a PC previously by decreasing the $d, pc$ number by 1, and delete the entry from their NTD table. Since $A_1$ was the primary replica for this RPC, it also has to use its $handler$ field to send an OK message to the RTM. The RTM then can tell the client program that the transaction is upgraded to a commit state. Figure 6 shows the actions of this scenario.

If a second transaction $T' = \{c_3 (p_2, d_1), c_4 (p_4, d_2)\}$ is submitted to the RTM before $A_2$ is recovered, then all feasibility checks will return PC states and the RTM will decide to do a partial commit. In that case, an entry in will be added to all replicas' NTD tables, with the new entry linked to the previous entry, as shown in Figure 5b. The $d, pc, i = 1, 2$ fields are increased by 1. Both transactions will be upgraded into the commit state when $A_2$ recovers.

Case 3: The network is partitioned into two parts. Suppose that we have the situation depicted in Figure 7, where the network is partitioned into two parts and a transaction is submitted to each part of the partition.

The feasibility checks of both partitions will return PC states and therefore both transactions will be partially committed by their RTMs. When the two partitions are reunited, both partitions will exchange information of their NTD tables and use Algorithm 5 to resolve conflicts. Since partition 1 has more partially committed replicas, $T$ will be committed in both partitions and $T'$ will be aborted in partition 2.

7.2 An application example

The system described in this paper has been used in our implementation to improve the reliability of a loosely integrated heterogeneous database system (23). This section describes the architecture of the system and some results of our experiments.
service offer from the trader and is given the agent's address (and the description of the service, of course). It then calls the agent and obtains the service.

Individual databases in the heterogeneous database system are allowed to maintain their autonomy, yet through the use of traders, they provide a substantial degree of information sharing. The trading manager executes on the same host as the database server. The trader, trading agents and user application programs can be executed on any Sun workstations.

The trader is implemented as a server that manages the trading context through RPCs. The trader is a key component of the system—the system cannot function once the trader fails and therefore it is replicated. That is, each location has a trader (and the replicated trading context) running on it. The trader on Location 1 (with an Oracle database running in that location) is assigned as the primary replica, whereas the other two traders are assigned as non-primary replicas. An RPC transaction manager (RTM) also runs on each location for managing RPC transactions initialized from the location. These RTM transaction managers have well-known addresses.

Our previous fault-tolerant RPC system [25] is used to implement the reliable communications required by various components of the RPC transaction management system. The RPC transaction features are now manually coded (using threads) and an extension of the interface definition file is under way to include the expression of RPC transactions.

In order to reduce the communication time, a communication channel (TCP stream) is established between the primary replicas and non-primary replicas (through the local RTMs) during the lifetime of an RPC transaction.

7.2.2. Experiments
We have carried out a number of experiments using the system. The first class of experiments is on the fault tolerance of the system, specifically the fault tolerance of the replicated traders. These experiments include the situations of (1) no failures, (2) a non-primary replica of the trader server fails and then recovers, (3) the primary replica of the trader fails and then recovers, (4) the combination of failures of a trader and an RPC transaction manager, and (5) the network is partitioned into two subnets (through software simulation) and then recovers. These experiments have shown that the system, when using algorithms described in this paper, can tolerate these failures.

The second class of experiments is on the performance of the replicated trader servers when there are no failures. As a comparison, we measured the performance of a null RPC, a single trader, two traders and three traders. Figure 9 depicts the architecture of these experiments.

All clients and servers were executed on Sun SPARC 5 workstations running the Solaris operating system. For Figures 9a and 9b, both the clients and server traders were executed on separate workstations within the same subnet. For Figures 9c and 9d, the client and the primary

FIGURE 8. Architecture of the loosely integrated system.

7.2.1. The architecture
Figure 8 depicts the architecture of our loosely integrated heterogeneous database system.

The system consists of the following components:

(i) Database servers. Each individual database server maintains database operations directed to it. Currently three database servers (Oracle, MiniSQL and Ingres) are running on three separate Sun workstations located about 100 km apart.

(ii) The trader. A trader is a third-party object that links clients and servers in a distributed system [24]. We use a trader to manage the shared information among participating databases of the loosely integrated database system. If a database system is willing to share part of its information with other database systems or outside applications, it exports (an update operation) that information as a service offer to the trader. These service offers are managed by the trader as trading context. Application programs (clients) wishing to make use of the shared information have to import (a read-only operation) such service offers from the trader and then access the database(s) concerned. Trading operations, including the export and import operations, are implemented as RPCs.

(iii) The trading manager. A trading manager is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results.

(iv) Trading agents. Trading agents are appointed by database servers to manage some special service offers. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then a trading agent can be appointed to manage the offer. A trading agent exports the service offer it manages to the trader. The client imports the

TABLE 1. Performance comparison (no failure).

<table>
<thead>
<tr>
<th>Service type</th>
<th>Mean (ms)</th>
<th>SSD (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Null RPC</td>
<td>3.07</td>
<td>0.51</td>
</tr>
<tr>
<td>Single trader (Read-only)</td>
<td>5.24</td>
<td>0.84</td>
</tr>
<tr>
<td>Single trader (Update)</td>
<td>5.31</td>
<td>0.87</td>
</tr>
<tr>
<td>Two traders (Read-only)</td>
<td>5.23</td>
<td>0.73</td>
</tr>
<tr>
<td>Two traders (Update)</td>
<td>6.14</td>
<td>1.06</td>
</tr>
<tr>
<td>Three traders (Read-only)</td>
<td>5.25</td>
<td>0.88</td>
</tr>
<tr>
<td>Three traders (Update)</td>
<td>6.27</td>
<td>1.04</td>
</tr>
</tbody>
</table>

(iii) The mean response time for an update operation has increased by 0.83 ms when the number of traders increases from one to two. This overhead is mainly due to the implementation of the algorithms for keeping the two replicas consistent. The increase is not significant since the communication between the two traders (a TCP channel, implemented through our own RPC system [25]) has been established before any update operation can be accepted by the replicas.

(iv) The mean response time for an update operation only increases by 0.13 ms as the number of traders increases from two to three. The main reason for such a small increase is due to the way we handle operations involving multiple replicas (as shown in Algorithms 1, 2 and 3), i.e. all these operations are sent to involved replicas concurrently (using threads). Since the traffic was low, the synchronization of these operations did not take too long.

Obviously, a different configuration will lead to a different result. For example, if each subnet is changed to a 100-Mbit Ethernet network, then the cost of communications among each subnet will become a significant part of the total cost since the speed of the ATM backbone is only 34 Mbits. However, it is also worth noting that a configuration with two replicas will perform similarly to a configuration with \( N \) replicas (\( N > 2 \) and being small) if the networks are equally fast. The main reason is that all replicas process and reply to requests from the primary replica in parallel.

The third class of experiments is on the impact of failures on the performance of the replicated traders. We carried out our experiments on the system of two traders, as depicted in Figure 9c, and measured the performance of export (update) and import (read-only) operations when the non-primary replica is faulty. In the case of the failure of the non-primary replica, RPC transactions can be treated as partial commit, and the primary replica has to record these partially committed transactions in its NTD table. Table 2 shows the result of the experiment.

It can be seen from Table 2 that the performance of read-only operations virtually does not change when the non-primary replica fails, since read-only operations are not affected in our system as long as there is one alive replica. However, the mean time of update operations has increased by 1.76 ms. This increase is mainly due to the disk I/O used to access the NTD table (which is stored as a disk file).

The last class of experiments is on the issues of recovery. We used the same system architecture (Figure 9c) as we did...
TABLE 3. Performance comparison (recovery).

<table>
<thead>
<tr>
<th>Recovery type</th>
<th>1 update (ms)</th>
<th>10 updates (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Mean</td>
<td>SSD</td>
</tr>
<tr>
<td>Non-primary failure</td>
<td>8.83</td>
<td>1.41</td>
</tr>
<tr>
<td>Partition (no conflict)</td>
<td>8.98</td>
<td>1.39</td>
</tr>
<tr>
<td>Partition (conflict)</td>
<td>12.35</td>
<td>1.56</td>
</tr>
</tbody>
</table>

In the third class of experiments, here we consider two types of recoveries: (1) a failed non-primary replica rejoins the system, and (2) a recovery from a network partition.

In the first case, during the time the non-primary replica is faulty, the primary replica may have carried out some update operations on the trading context. These update operations have been recorded in the NTD table of the primary replica and should be executed by the non-primary replica when it recovers. When the non-primary replica recovers, it cleans up its NTD table and sends a reuniting message to the primary replica. The primary replica will then send the relevant NTD table entries to the recovering replica and commit the outstanding partially committed transactions. In the meantime, the non-primary replica will carry out the missed operations. Here, we measured the time from the point that the recovering replica sends out the reuniting message to the time that it has completed all missed operations.

In the second case, when the network is partitioned, the two replicas are isolated from each other (we simulated this failure by deliberately coding wrong addresses into the two replicas). According to our algorithms, both replicas will become the primary replica of their own replica and will be able to accept transactions. We have considered two situations for the update operations carried out during the network partition: (1) all of these update operations are not conflict, and (2) all of them are conflict.

When the two replicas recover from the network partition failure, they have to exchange their NTD tables and use Algorithm 5 to resolve conflicts. For situation (1), since there is no conflict, both replicas will execute the missed operations and commit their own partially committed operations. For situation (2), since the original primary replica (Trader 1) has a smaller sequence number, the original non-primary replica (Trader 2) has to abort all its partially committed operations and execute the operations carried out by Trader 1 according to Algorithm 5. Here, we measured the time from the point that Trader 2 issues the reuniting message to the time that it has completed all missed operations.

In all cases of the experiments related to recovery issues, the numbers of update operations during the failure time were set to 1 and 10. Table 3 lists the test results.

We make the following observations from Table 3:

(i) The difference between the time used to recover from a replica failure and the time for a network partition failure is small when there is no conflict. This is mainly because the same algorithms were used and the recovering replica Trader 2 (the one where we measured the time) had to perform the same operations in both cases.

(ii) The mean recovery time increases significantly as the number of update operations increases. This is mainly due to the way that we process update operations: the update operations were processed sequentially on the trading context file. We have proposed an algorithm to parallelize recovery operations [26], and it is planned to embed the algorithm into the RPC transaction system.

(iii) The mean recovery time increases dramatically as the number of conflict operations increases. This is mainly due to the way we process conflict operations. For each conflict operation, a replica has to perform two disk I/O operations (one on the NTD table and one on the trading context file) sequentially. Once again, our proposed algorithm for parallelizing recovery operations [26] can play an important role here in improving the performance.

8. REMARKS

A system for building reliable computing over an RPC system is described in this paper. The system combines the replication and transaction techniques together and embeds these techniques into the RPC system. The paper describes the models for replicas, RPCs, transactions, and the algorithms for managing transactions, replicas, and resolving conflicts during system recovery. Finally, an informal correctness analysis is carried out and an illustration example and an application example are described.

Although many ideas used in this paper are well known, and some of them have been implemented in commercial products, the paper does provide the following novel contributions. The first major contribution of the paper is the combination of replication, transaction management and RPC techniques to form a system that supports the development of reliable services in the RPC level. The main advantages of supporting fault tolerance in the (lower) RPC level instead of the (higher) application/service level are: (1) efficiency, since failures can be dealt with in a lower protocol level, (2) failure containment, since failures can be dealt with quickly and can reduce the danger of failure propagation, and (3) lower development cost, since the lower-level support means that many higher-level applications/services will be less likely to repeatedly develop their own fault-tolerant protocols.

The second major contribution of the paper is the introduction of a partial commit (PC) concept to facilitate the processing of transactions in case of failures. The motivation of using the PC state for a transaction is to let the transaction proceed even if a replica is down or the network is partitioned. Partially committed transactions will be upgraded to commit states or downgraded to abort states.
when the system recovers from the failure that affected the transactions.

A similar work of combining replication, transaction management and RPC techniques to support fault-tolerant computing is the Encina toolkit [27, 28]. Encina extends the basic DCE services to include facilities such as the transactional RPC and the Encina Monitor (a transaction processing monitor). The major difference between our work and the Encina toolkit is that Encina only has two states for a transaction: commit or abort. Our proposal uses a partial commit concept to let transactions proceed even during a replica failure or a network partition.

The fulfillment transaction approach [7] also allows transactions to proceed during a network partition, by generating some new transactions (fulfillment transactions) that will be processed when the network partition failure is recovered. The main difference between our work and the fulfillment transaction approach is that our model does not require the application-specific knowledge to build those fulfillment transactions. That gives us a freedom of implementing our model in a lower level of the system hierarchy which can be shared by many different applications. With enough business policies embedded into the generation of fulfillment transactions, the chance of manual intervention in these fulfillment transactions can be greatly reduced. However, our method avoids the use of any manual intervention.

The algorithms described in the paper have been used in improving the reliability of traders used in an experimental loosely integrated database system. A series of experiments has been conducted on the implementation to test the reliability of the system, the overhead for maintaining replicas, the impact of failures on the system performance, and the overhead during recovery. From these experiments we have shown that the proposed algorithms can provide reliability without incurring too much negative impact on the system performance.

ACKNOWLEDGEMENTS

The authors would like to thank the referees for the helpful comments.

REFERENCES

BUILDING RELIABLE PROGRAMS IN A REPLICALED ENVIRONMENT USING REMOTE PROCEDURE CALLS

W. Zhou and A. Gosclinski*  

Abstract

This paper addresses the problem of building reliable computing programs over remote procedure call (RPC) systems by using replication and transaction techniques. We first establish the computational model: the RPC transaction model. Based on this RPC transaction model, we present the design of our system for managing RPC transactions in the replicated-server environment. Finally we present some results of a correctness study on the system and two examples of the system.

Key Words

Fault-tolerant computing, distributed systems, remote procedure call, transaction management, replication, distributed database

1. Introduction

Two approaches are commonly used in supporting fault-tolerant computing. The first approach provides programming languages that are targeted at developing fault-tolerant systems. Typical examples of this approach are the Ada 95 [1], the Fault-Tolerant Concurrent C [2], and the fault-tolerant version of the SR language [3]. The second approach provides a fault-tolerant computing toolkit or a model that can be used together with general programming languages and standard operating systems. Typical examples of this approach are the ISIS toolkit [4], the ARGUS system [5], the location-based replication paradigm [6], and the RPC transaction management system [7]. This paper follows the second approach.

Remote procedure call (RPC) is perhaps the most popular model used in today’s distributed software development and has become a de facto standard for distributed computing. The use of RPC facilitates the building of distributed programs and eliminates concerns for the communication mechanisms from the programs that use remote procedures. Only fundamental difficulties of building distributed systems such as synchronisation and independent failure of components are left in RPC programming.

Many leading computer companies have agreed on a vendor-neutral distributed computing environment (DCE) architecture proposed by the Open Software Foundation [8]. This architecture is designed under the client/server model, and requires the interactions between its components to follow the RPC paradigm. Although the DCE ar-
performance penalties incurred by using process groups.

The location-based paradigm for replication proposed by Triantafillou and Taylor [6] addresses the problem of combining reliability with performance issues. It uses replication and transaction techniques to develop reliable programs. The proposal provides reliability similar to quorum-based replication protocols, but with transaction delay similar to a one-copy system. However, the proposal cannot adequately deal with network partitions.

The RPC transaction management system proposed by Zhou and Molnar [7] uses RPC and transaction techniques to develop reliable programs. RPCs are grouped into transactions that are guaranteed to be atomic. However, the proposed system does not work in a replicated environment.

The Coda file system [12] supports the replication of file volumes, managing the resulting multiple-update problem even in the presence of server failures. The replication of file volumes produces a fault-tolerant service. The most successful feature of the Coda system is its support for disconnected operations. However, the reintegration of files after a disconnection involves manual intervention in the case of conflicting updates.

The purpose of this paper is to combine replication, transaction management, and RPC together to form a reliable and efficient distributed computing environment. Fault-tolerant programs developed in our environment should be able to tolerate single failures such as a server failure, a site failure, or even a network partition without involving manual intervention. These programs will also have efficiency similar to a non-replicated system when there is no component failures.

The remainder of the paper is organized as follows: Section 2 introduces the replicas, the RPC model, the transaction model, and the failure semantics. Section 3 presents the model and the algorithm for transaction management. Section 4 describes the replica management. Section 5 discusses the conflict-resolution algorithms. Section 6 presents some results of a correctness study of the proposed system. Section 7 describes an illustration example and an application example of the system. Section 8 concludes the paper.

2.1 System Models

2.1 Replicas

A distributed system with replicated servers consists of many sites interconnected by a communication network. A service is provided by a group of replicated servers (called replicas) executing on some sites. These replicas manage some common data objects that can be shared by many clients. For simplicity, we assume that each replica knows the location of other replicas that store the same data objects. This assumption can be loosened if a replication directory service is used.

When requesting a service, a client specifies the service through, say, an attributed name [13]. The exact location of the service and the server that provides such a service will be determined by the system.

We model a service as a set of replicas: \( S = \{ S_1, S_2, \ldots, S_i, \ldots, S_n \} \), where \( i = 1, 2, \ldots, n \) are called the sequence numbers of these replicas. Each replica \( S_i \) manages a set of data objects: \( O_i = \{ O_1, O_2, \ldots, O_i \} \). The consistency constraint requires that for \( i, j = 1, 2, \ldots, n \) and for each service \( S \), \( O_i \equiv O_j \).

2.2 The RPC Model

Each replica in our system provides a number of remote procedures that can be called by clients for processing the data objects managed by the replica. We use \( P \) to denote the set of all remote procedures provided by all replicas of the system:

\[ P = \{ p \mid p \text{ is a remote procedure provided by the system} \} \]

After we make an RPC, the call may return successfully or fail. There may be several reasons for the failure of a call, such as "object not free", "server error", or "communication error." With some error conditions, it is clear to the client that the procedure was not executed, while with other error conditions, it is not clear if the procedure was executed or not. For example, on receiving an "object not free" or a "server error," we are sure that the RPC was not performed. But on receiving a "communication error," the client will not be able to tell if the call was performed or not, because we do not know whether the error happened before or after the calling request arrived at the destination host.

We define the effects of an RPC as the processing of one data item of the object system. Hence we can abstract an RPC as a mapping of the following type:

\[ e : P \times O \rightarrow \{ OK, FL, US \} \]

where \( O \) is the union of all data objects managed by all services of the system. The values of the target set have the following meanings:

- **OK**: This means that no failure occurred during the RPC's execution. By \( e(p, d) = OK \), where \( p \in P \) is a remote procedure and \( d \in O \) is the data object processed by the call, we mean that the RPC was successful (i.e., the remote procedure performed the job).

- **FL**: This means accessing failure. By \( e(p, d) = FL \), we mean that the destination server (the server that exports the remote procedure \( p \)) could not perform the job because, for example, the arguments between the client and server do not match, the versions are different, or the object managed by the server is not free. This means that the RPC has not executed.

- **US**: This means unknown state. By \( e(p, d) = US \), we mean that the client cannot tell if the RPC has or has not been executed because, for example, the destination host (the host that the server exporting procedure \( p \) resides on) is down, or the server is down, or the links between the client and server are down (that is, the client host and the server host belong to two partitioned sub-networks). In that case the RPC request may be lost, or the return message may be lost. So we do not know if
the remote procedure has been executed or not.

Without loss of generality we assume that all RPCs are update-oriented operations. That is, if \( c(p, d) \) is successful, it transforms the data object \( d \) from the existing state to a new state.

### 2.3 RPC Transaction Model

We define a (parallel) RPC transaction as \( T = (c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_n(p_n, d_n)) \), where \( c_i(p, d) \) is an RPC, and \( p_i \in P, d_i \in D \). The semantics of an RPC transaction are that after issuing the transaction, all \( c_i \) of \( T \) will be executed if no error occurs (commit or OK), or if any one of them fails, all executed RPCs will be rolled back (abort or FL). An RPC \( c_i(p_i, d_i) \) in \( T \) returns OK if, and only if, all replicas of \( d_i \) have successfully performed the procedure \( p_i \). Similarly, \( c_i(p_i, d_i) \) returns FL if, and only if, some of its replicas of \( d_i \) failed to execute the procedure \( p_i \) on \( d_i \).

In addition to the two normal states (commit and abort), we define a partial commit (PC) state. The real meaning behind the PC state is that a replica is not accessible (e.g., the replica is down or the network is partitioned), but there are other replicas that can provide the same service. So the transaction has only performed RPCs on all replicas that are alive (can be reached now). For those replicas that are not accessible now, the transaction effect will be resolved when these replicas re-join the service (e.g., the failed replica recovers or the network is re-united). During the conflict-resolution phase, all the transactions that returned PC will be checked. If the work of such a transaction does not conflict with any other transactions (e.g., it does not use any common data objects with other transactions), then it is considered to be safe and will be committed. If a conflict is detected between two transactions, then one of them is chosen as a victim (un-safe) and will be aborted. The other will then be safe and committed. This is done asynchronously.

The execution of all \( c_i \) in \( T \) is in parallel. Some parallel primitives can be built for parallel execution of remote procedures [14]. Sequentially executed transaction can be easily established from the parallel model. However, if operations within a transaction are to be executed sequentially, the serializability [15] must be considered. In this paper, we only consider parallel RPC transactions. Fig. 1 indicates the semantics of an RPC transaction.

![Figure 1. Semantics of the RPC transaction.](image)

If we do not need to distinguish individual RPCs within an RPC transaction \( T \), we then write \( T = \{c(p, d)\} \) and use \( |T| \) to represent the number of parallel RPCs within \( T \).

### 2.4 Failures

There are four classes of failures in a replicated-server environment executing RPC transactions: transaction abort, replica-is-down, site-is-down, and network partition.

An RPC transaction \( T \) aborts if any of its RPC returns an FL (e.g., the requested data object is not free, or if all replicas of a service that the RPC accesses are down), or \( T \) was in a PC state and then the conflict-resolution process identifies that \( T \) is un-safe. The usual treatment for a transaction failure is to retry it after a random period of delay. The random delay is necessary to avoid oscillation when two or more transactions need to use the same data object at the same time.

A replica is down means that the replica is not accessible. In this case, other replicas of the same service should still provide the service to clients (an RPC returns FL if all replicas of a service that it accesses are down). However, when the failed replica recovers and re-joins the service, it may have stale information because some data objects may have been updated when the replica was down. To solve this problem, we set all transactions on the live replicas of the service to the PC state. These partially committed transactions will be made to commit when the failed replica re-joins the service and starts the recovery process. At the same time, the failed replica will update all its stale information.

A site is down means that the replica(s) that runs (run) on it is (are) down. We assume that each site has only one replica running on it. That reduces the problem into a replica-is-down failure.

A network partition failure means that replicas of a service may belong to two disconnected partitions. In this case, the two sets of replicas should still provide the service to clients. However, when the two partitions are re-united, the two sets of replicas may have performed some conflicting operations. To solve this problem, we also set all transactions performed during the network partition on these two parts of replicas to PC states. These partially committed transactions will be resolved when the two parts are re-united.

We use "system failures" to denote the latter three failures and we assume a single failure for system failures. That is, at any time, only one of the replica-is-down failure, or the site-is-down failure, or the network-partition failure occurs. As discussed above, we can actually classify them into two types of failures: the replica failure and the partition. A replica failure means that a replica is down or the site that the replica is running on is down. A partition means the network is partitioned into two disconnected sets and both parts have some replicas running on them. In both cases, we can divide the replicas into two parts. For simplicity, we only assume crash failures in this paper. Byzantine failures can also be dealt with when the number of replicas of a service is more than three and
a majority voting scheme is used.

3. RPC Transaction Management

3.1 The RPC Transaction Processing Model

We define a primary replica for a data object \(d\) as a replica that is the best (e.g., the nearest site to the RPC transaction manager (described below) that accepts the client request, however, the measure is left for individual applications) in performing an RPC \(c(p, d)\). Any replica can be chosen as the primary replica for a particular RPC. The management of primary replica information is itself a difficult issue. We will concentrate on the transaction management and will therefore ignore this problem here.

Three system components are involved in processing a transaction submitted by a client. 

* An RPC transaction manager (RTM) accepts the transaction \(T(c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_n(p_n, d_n))\) from the client. The RTM sends each RPC \(c(p_i, d_i)\) to a primary replica of \(d_i\) and asks the primary-replica to check if the RPC can be performed or not. We denote this operation as \(c(p_i, d_i)\). The RTM then acts as a coordinator for managing the atomicity of \(T\) through the help of the primary replicas. Section 3.2 describes the RTM algorithm.

* A primary replica accepts, from the RTM, an RPC \(c(p_i, d_i)\) and the request to check the executability of the RPC (the \(c(p_i, d_i)\) operation). The primary replica sends the RPC to all replicas of data object \(d_i\) and asks all replicas (including itself) to check if they can execute the RPC (e.g., if \(d_i\) is free). We use \(c(p_i, d_i)\) to represent this operation. The primary replica acts as a coordinator for managing the RPC \(c(p_i, d_i)\) to be performed on all replicas (including itself). Section 4.1 describes the coordinating algorithm.

* Each replica of the data object \(d_i\) accepts, from the primary replica of \(d_i\), the RPC \(c(p_i, d_i)\) and the request to check the executability of the RPC (the \(c(p_i, d_i)\) operation). The replica then cooperates with the primary replica by returning the executability check and performing the RPC when requested. Section 4.2 describes the cooperating algorithm.

The actual effect of an \(c(p_i, d_i)\) call depends on the associated \(h(c(p_i, d_i))\) calls. That is, if all \(h(c(p_i, d_i))\) calls return OK, \(c(p_i, d_i)\) returns OK. If any \(h(c(p_i, d_i))\) call returns FL, \(c(p_i, d_i)\) also returns FL. If no \(h(c(p_i, d_i))\) call returns FL and there are PC returned by \(h(c(p_i, d_i))\) calls, or no PC return but there are US or OK returns (the primary replica must return OK or PC in this case), then \(c(p_i, d_i)\) returns PC. An \(c(p_i, d_i)\) may also return US if the primary replica for \(d_i\) is not accessible.

A client simply submits a transaction request to the RTM and waits for the transaction result and the resulting state. The resulting state returned at this stage can be one of OK, FL or PC. As soon as the client receives a resulting state of OK or FL, it can continue with its next operation. That means, other operations, such as operations required for consistency, or for continuous committing will be dealt with by the system without the participation of the client.

However, if no primary replica returns an FL but there are some primary replicas return US (i.e., those replicas are not accessible), then the RTM will retry to find new primary replicas for those primary replicas who returned US. This process repeats until either there is no more available replicas that can act as primary replicas for some particular data objects, or the newly found primary replicas return OK/PC/FL.

If the transaction returns a PC state, it should enable an asynchronous receive procedure (similar to an exception handler) to process the following asynchronous message returned for the transaction. This message will come when the partially committed transaction is resolved. In this case, either a commit or an abort message will be returned. The asynchronous receive procedure should then process this accordingly. Figure 2 depicts the model for RPC transaction processing.

![Figure 2. RPC transaction processing](image)

3.2 The Transaction Manager

The transaction manager has three basic functions to perform when it receives a transaction \(T\).

* It submits operations to their primary replicas. The manager is responsible for locating primary replicas for each RPC and then sending these RPCs to their corresponding primary replicas.

* It manages the atomicity of the transaction \(T\). It uses the 2-phase-commit (2PC) protocol to ensure that all RPCs of \(T\) to the primary replicas return OK, T asks all primary replicas to commit and returns OK. If any of these RPCs return FL, it asks all primary replicas to abort and returns FL. In this case, any uncommitted update will eventually be rolled back. If no operation returns FL, but there is some operations return US, then the manager will find new primary replicas for those who returned US and try again. If all operations return OK or PC (and there is at least one PC return), then T asks all primary replicas to partially commit and returns PC.

* It sets up an asynchronous receive procedure if a transaction returns a PC. The procedure awaits messages from all primary replicas that returned PC.
all of these primary replicas later return OK, then the procedure will up-grade the transaction to a commit state. If any of the primary replica later returns an FL, then the procedure will down-grade the transaction to an abort.

We define a function locatePrimaryReplica() which takes as input an RPC and returns the primary replica location of the RPC. If the function returns a NULL, then this means that there is no more available replica that can be a primary replica for the RPC concerned (e.g., all replicas for processing a particular object are cut off from the transaction manager).

In this case the transaction has to be aborted. This function also guarantees that if $r_1(p, d_i), r_2(p, d_j) \in T$ and $d_i = d_j$, then locatePrimaryReplica($r_1(p, d_i), r_2(p, d_j)$) = locatePrimaryReplica($r_2(p, d_j)$). That means that the two RPCs of the same transaction will use the same primary replica if they access the same data object.

The algorithm manage_rpc_transaction() of Listing 1 implements the RTM. The constant MAXRFRCS defines the maximum number of RPCs allowed in an RPC transaction.

Listing 1. Algorithm 1 — The RTM Algorithm.

```
morgan_rpc_transaction() {
    primary address handle primary address[MAXRFRCS];
    initComplete := 0;
    while (TRUE) {
        if receive RPC, then process pending until a transaction comes over
        if many transactions come, they are queued until the processing of the current transaction is over
        if receive $T = \{ t_1, \ldots, t_n \}$;
            while (initComplete = 0) {
                /* concurrently locate the primary replicas of each RPC */
                for each primary address handle primary address[p] {
                    if (primary address[p] = locatePrimaryReplica(p)) {
                        if (initComplete = 0) {
                            /* concurrently set all primary replicas to abort */
                            C_OBJECT(p) = "aborting",
                            y[p], d[p] \in T;
                            /* concurrently execute the RPCs through the help of primary replicas */
                            C_OBJECT = \{ t_1, \ldots, t_n \};
                            break;
                        }
                    }
                }
                /* concurrently set all primary replicas to commit */
                for each primary address primary address[p] {
                    y[p], d[p] \in T;
                    switch case 
                        S(\{ x[p] \}) \in S \in T;
                        /* all x[p], d[p] executed and return OK */
                        for each primary address primary address[p] {
                            /* concurrently set all primary replicas to commit */
                            C_OBJECT(p) = "committing",
                            y[p], d[p] \in T;
                            /* get the request from the transaction returns OK */
                            if initComplete = 0;
                            break;
                        }
                    case S(\{ y[p] \}) = 0;
                        if (initComplete = 0) {
                            /* concurrently set all primary replicas to abort */
                            C_OBJECT(p) = "aborting",
                            y[p], d[p] \in T;
                            /* get the request from the transaction returns FL */
                            if initComplete = 0;
                            break;
                        }
                    case S(\{ x[p] \} = \emptyset \& S \in T)
                        if (initComplete = 0) {
                            /* concurrently set all primary replicas to commit */
                            C_OBJECT(p) = "committing",
                            y[p], d[p] \in T;
                            /* get the request from the transaction returns OK */
                            if initComplete = 0;
                            break;
                        }
                    case S(\{ y[p] \} = \emptyset \& S \in T)
                        if (initComplete = 0) {
                            /* concurrently set all primary replicas to abort */
                            C_OBJECT(p) = "aborting",
                            y[p], d[p] \in T;
                            /* get the request from the transaction returns FL */
                            if initComplete = 0;
                            break;
                        }
                }
            }
            break;
        }
    }
}
```

4. Replica Management

4.1 The Coordinating Algorithm for the Primary Replicas

When a primary replica receives an RPC request $x(p, d)$ from a transaction manager, it uses the coordinating algorithm to maintain the consistency of all replicas in terms of the RPC. This section describes the coordinating algorithm.

In the coordinating algorithm, the primary replica uses the 2PC protocol to ensure replication consistency. In the first phase, the primary replica asks all replicas (including itself) to check the executability of the RPC (i.e., the $X(x(p, d))$ operation). If all replicas return OK for such execution, the primary replica returns OK to the transaction manager.

If the transaction manager requests commit, then in the second phase the primary replica asks all replicas to commit the RPC execution. If any replica returns FL, then the primary replica returns FL to the transaction manager and asks all replicas to abort the operation in the second phase. If all operations return either OK or FL (in this case, the primary replica must return OK or FL and other non-primary replica may return OK, PC or US), the primary replica returns PC to the transaction manager. The primary replica also records the number of replicas that return OK and PC and the smallest sequence number among these replicas. These two numbers will be used in conflict resolution. If the transaction manager requests partial commit in the second phase, the primary replica asks all replicas to partially commit.

If the transaction manager requests an abort, the primary replica then asks all replicas to abort, no matter what was returned by the primary replica in the first phase. The coordinating algorithm is listed in Listing 2. We assume that $S_i$ is the primary replica.


```
primary_replica, perform() {
    init, DoRPCs, insertRPC;
    if (TRUE) {
        /* receive RPC from RTM, the process suspends until an RPC comes over */
        if (more RPCs come, they are queued up until the current RPC is done */
        if (start the first phase)
            \$d = \$d \in \{ 0, 1, \ldots, S \}$
        else
            $S_i = \emptyset$ be the primary replica:
            /* concurrently ask replicas $S_i = \{ 1, 2, \ldots, 4 \}$ to check the executability of the RPC */
            C_OBJECT = [1, 2, 3, 4];
            break;
        end;
    end;
}
```

```
switch {  
case \( R_{\text{OK}}, \neq \emptyset \cap (R_{\text{OK}} \neq R_{\text{RC}} \neq R_{\text{RC}} = \emptyset) \):  
  /* all RPCs (p,d) executed and returned OK */  
  tell the RIM that the first phase returns OK;  
  break;  
case \( R_{\text{OK}} \neq \emptyset \):  
  /* at least one replica returns FL */  
  tell the RIM that the first phase returns FL;  
  if \( \#(\text{RPCs}) = |R_{\text{OK}}| \) AND \( \text{smallestPC} = \text{min}(\text{RPC}(p,d)) \in R_{\text{OK}}) \)  
  break;  
case \( R_{\text{OK}} \neq \emptyset \cap (R_{\text{OK}} \neq R_{\text{RC}} \neq \emptyset) \):  
  /* partial commit */  
  tell the RIM that the first phase returns PC;  
  \( \text{RPCs} = |R_{\text{OK}}| + |R_{\text{RC}}|; \) \( \text{smallestPC} = \text{min}(\text{RPC}(p,d)) \in R_{\text{RC}} \cup R_{\text{OK}}) \)  
  break;  
} /* switch */  
/* second phase now. Receive the RIM's second phase command */  
receiveRIM(command);  
switch {  
case command is "commit":  
  /* all replicas to commit */  
  \( \text{GOBEGIN} \);  
  \( \text{send}(\text{RPC}(\text{commit})), i = 1, 2, ..., k; \)  
  \( \text{END;} \);  
  break;  
case command is "partial commit":  
  /* all replicas to partially commit */  
  \( \text{GOBEGIN} \);  
  \( \text{send}(\text{RPC}(\text{partial commit})), i = 1, 2, ..., k; \)  
  \( \text{END;} \);  
  break;  
case command is "abort":  
  /* all replicas to abort the work */  
  \( \text{GOBEGIN} \);  
  \( \text{send}(\text{RPC}(\text{abort})), i = 1, 2, ..., k; \)  
  \( \text{END;} \);  
  break;  
} /* switch */  
} /* while */  

In addition to the coordination work, the primary replica has to do all the work of a non-primary replica. The next section describes the role performed by a non-primary replica.

4.2 The Cooperating Algorithm for All Replicas

When a non-primary replica receives a request from a primary replica, it checks whether the request can be proceeded or not and acts accordingly.

- If the request can be performed, the non-primary replica locks the required data object and returns OK. Later, if the primary replica asks to commit the operation, the non-primary replica performs the operation and releases the lock. If the primary replica asks to abort the operation, the non-primary replica releases the lock. If the primary replica asks to partially commit the operation, then the non-primary replica partially commits the operation and records this event.

- If the non-primary replica finds that the operation cannot be executed (e.g., the required data object is not free), it then returns an FL.

- If the non-primary replica finds that the data item is already in partially committed state, then it returns PC to the primary replica. Later if the primary replica asks to partially commit the operation, the non-primary replicas then partially commits the operation and records the event. If the primary replica asks to abort the operation, then the non-primary replica aborts the operation.

The 2-phase-commit protocol used by the RPC transaction manager guarantees that the replicas will stay in a consistent state if a transaction returns an OK or an FL.

However, to guarantee all replicas do the same thing to the transactions with PC pending, a consensus should be reached among all replicas. This is a classic consensus problem in fault-tolerant computing [16]. We use a Need-To-Do (NTD) table to record the events of partially committed RPCs. Then during a recovery, we let replicas exchange their NTD table entries in order to reach a consensus for actions on the transactions with PC returns. The NTD table of each replica is kept in stable storage [17]. Therefore, information stored in the NTD table will not be affected by system failures. The NTD table structure is listed in Listing 3.

Listing 3. The Need-To-Do Table.

<table>
<thead>
<tr>
<th>Data Structure for the NTD Table</th>
</tr>
</thead>
</table>

When the primary replica asks for a partial commit for an RPC, all replicas (including the primary replica) will record this event into its own NTD table as a new entry. If the NTD is such an entry, then t_rpc contains the name of the RPC, t_data contains the data object used in the RPC, t_pc stores the number of replicas that have partially committed this RPC, and t_sm stores the smallest sequence number among all the replicas that have partially committed the RPC. These two numbers (t_rpc and t_sm) are sent by the primary replica to each replica when it asks for partial commit (see Algorithm 2). They are used later by the conflict resolution algorithm to determine which partial commit should be up-graded to a commit and which partial commit should be down-graded to an abort if a conflict occurs.

In order to down-grade a PC to an abort, it must be the image of the data object kept in the NTD table. t_sm is used to record the address of the image before the image.

The order of each partially committed RPC over a data object is also very important when a recovery replica carries out these RPCs. We use a pair of pointers to record this order. The t_rpc stores the previous partially committed RPC and the t_data stores the next partially committed RPC for the same data object. The t_handler stores the address of the asynchronous receive handler if the replica is the primary replica.

We associate with each data object d a lock d.lock and a partial commit flag d.pc. The actual effect of \( k((p,d)) \) on d is to check or change the values of d.lock and d.pc. That is, if d.lock = -1 (locked), \( k((p,d)) \) returns FL. If d.lock = d.pc = 0 (free and not partially committed), \( k((p,d)) \) returns OK and sets d.lock = -1. If \( d.lock = 0 \) AND (d.pc ≥ n + 1) (free but partially committed), \( k((p,d)) \) returns PC and sets d.lock = -1 and d.pc = n + 1. If the replica is down or the network is partitioned during the operation (i.e., the replica is un-
reachable), a US is returned by the RPC system.

All RPCs of a transaction are actually performed in the second phase of the algorithm. During this phase, each replica has to release the lock d.lock (set to 0) if the c(p,d)
has a normal commit or a partial commit. An entry about this partial commit is also inserted into the NTD table.

The conflict resolution algorithms are then responsible for upgrading this to a normal commit or for downgrading this to an abort. The cooperating algorithm is given in Listing 4.

Listing 4. Algorithm 3 — The Cooperating Algorithm.

```c
all_replay, process()

1. NTD E:
2. begin CMS:
3. /* receive RPCs from primary replicas, the process suspends until an RPC enters */
4. while (TRUE) {
5.  /* more RPCs come, they are queued up until the current RPC is done */
6.  receive(rPC, c(p,d), d.lock = 1; mState = OK; /* lock it */
7.  tell the primary replica that the first phase returns OK;
8.  break;
9.  case (d.lock = 1; pc = 0):
10.    /* the data object is free and not partially committed */
11.    d.lock = 1; mState = OK; /* lock it */
12.    tell the primary replica that the first phase returns OK;
13.    break;
14.  case (d.lock = 1; pc = 1; mState = PC):
15.    /* lock it; increase partial commit No */
16.    d.lock = 1; pc = 1; mState = PC; /* lock it */
17.    tell the primary replica that the first phase returns PC;
18.    break;
19.  case (d.lock = 1; pc = 0; mState = OK;)
20.    /* the US state is returned by the RPC system if a replica is unreachable */
21.    /* second phase */
22.    receive(rPC, c(p,d), d.lock = 0; mState = OK;)
23.    do (retry a(c,p,d)) until retry OK;
24.    d.lock = 0; /* lock the data object */
25.    break;
26.  case (command is "commit"): do (retry a(c,p,d)) until retry OK;
27.    /* the commit procedure is executed */
28.    break;
29.  case (command is "partial commit"): do (retry a(c,p,d)) until retry OK;
30.    /* the commit procedure is executed */
31.    break;
32.  case (command is "abandon"): do (retry a(c,p,d)) until retry OK;
33.    break;
34.  } /* switch */
35. } /* while */
```

5. Conflict Resolution

An RPC returns US only when the server is down or the server is unreachable (e.g., the network is partitioned). If the server is down, it will eventually be repaired and return to service. In that case, the server has missed all updates to its data objects while it is down. Fortunately, we have set these updates in "partial commit" state and recorded them in NTD tables. Therefore, the conflict resolution algorithms can use this information and make the data objects managed by this failed replica in line with all other replicas. If the network is partitioned into two disconnecting parts, the two parts will eventually be re-united again. In this case, replicas in both partitions may have some partially committed updates. The conflict resolution algorithms are also responsible for making replicas in these two parts consistent.

Since the recovery process is very critical, we assume that no system failures will occur during the recovery process. This assumption can be loosened if more communication and stable storage are assumed.

When recovering from a replica failure, we assume that there is a process that deletes all entries of the NTD table of the recovering replica, except those entries with t.handler ≠ Ø. The reasons for doing this are: (a) those entries with t.handler ≠ Ø are no longer useful — they will be dealt with by other alive replicas; (b) those entries with t.handler ≠ Ø have to be dealt with by the recovering replica because it was the primary replica for the particular RPC. This NTD table is sent to all alive replicas by the recovering process.

Two algorithms are needed during the recovery of a system failure. When recovering from a replica failure, the recovering replica has to send a "re-uniting" message (it includes the sequence number of the recovering replica) to other alive replicas. This enables the alive replicas to send outstanding NTD table entries to the recovering replicas by using the send_out.ntd() algorithm.

When recovering from a network partition, replicas of each part of the partition have to send a "re-uniting" message (it includes the sequence numbers of all replicas of the partition) to replicas of the other part. This enables the exchange of NTD table among replicas in both parts by using the send_out.ntd() algorithm.

5. Conflict Resolution

List 5. Algorithm 4 — The Algorithm for Sending Out an NTD Table.

```c
send_out.ntd()

1. while (TRUE) {
2.  /* detect the re-uniting messages */
3.  /* the process suspends until such a message comes */
4.  msgList contains the sequence numbers of the re-uniting messages;
5.  send(rPC, c(p,d), msgList, NTD, i ∈ msgList);
6.  COROID;
7.  /* while */
```
The conflict resolution() algorithm is used by all replicas to resolve potential conflicts for a data object. The algorithm receives NTD entries from replicas of the other part. Only the first arriving NTD table is accepted by the algorithm, and others are ignored. The algorithm then checks the received NTD entries against its own NTD entries to resolve potential conflicts.

We define a leader of an NTD table as the first partial commit entry for a data object d. That is if t ∈ T is a leader in an NTD table T, then t.previous = ∅. The algorithm uses three functions for processing NTD table entries led by a leader:

- The abort.all() function aborts all RPCs led by the leader q of the NTD table t_m.
- The commit.all.own() function commits all RPCs led by the leader q of the NTD table t_m.
- The commit.all.received() function commits all RPCs led by the leader of the NTD table t_m.

We first describe these three functions:

Listing 6. The abort.all() function. It finds the tail of the entries led by q and rolls back all RPCs from the tail to the leader. If the replica is also the primary replica of the RPC, an FL message is also sent to the asynchronous receive procedure.

Listing 7. The commit.all.own() function. It finds the tail of the entries led by q and commits all RPCs from the tail to the leader. If the replica is also the primary replica of the RPC, an OK message is also sent to the asynchronous receive procedure.

Listing 8. The commit.all.received() function. It finds the tail of the entries led by q and commits all RPCs from the tail to the leader.


6. Correctness

In this section we outline the informal analysis of the correctness. We first assume the life-cycle of the system entities (sites, links, and replicas) is:

work — crash — repair and restart — work

Without loss of generality, we assume that the maximum down time (including crash, repair and restart time) is finite, and is denoted by T_d.

Assertion 1: If a transaction returns OK, all its RPCs have been executed successfully.

Proof: The only way that a transaction returns OK is that in Algorithm 1 we have (ROK = ∅ ∧ SOK = ∅). This means all RPCs to the primary replica have returned OK. If a primary replica returns OK if, and only if, in Algorithm 2, we again have (ROK = ∅ ∧ ROK = ∅). This means all RPCs to all replicas return OK. A replica returns OK if, and only if, in Algorithm 3 the data object involved is free and is not partially committed. If the transaction returns OK, in the second phase of Algorithm 1 we order all primary replicas to commit the transaction. In this case, all primary replicas will order their replicas to commit in the second phase of Algorithm 2 and, therefore, all replicas will successfully perform the real RPC in their second
phase of Algorithm 3.

Assertion 3: If a transaction returns FL, no RPCs of the transaction have been executed.

Proof: There are two ways that a transaction could return an FL. The first way that a transaction returns FL is that in Algorithm 1 we have \( R_{FL} \neq \emptyset \). This means some of the primary replicas returned FL. A primary replica returns FL if, and only if, it is not free. If the transaction returns FL is in the second phase of Algorithm 1 we order all primary replicas to abort the transaction. In this case, all primary replicas will order their replicas to abort in the second phase of Algorithm 2 and, therefore, in Algorithm 3, those replicas that returned OK will release their locks to the data object. The replicas that returned PC will set the associated integer of the data object to the free state. The replicas that returned FL will do nothing. In any case, none of the RPCs is executed.

The second way that a transaction returns an FL is that in Algorithm 1 we cannot find any more replicas to be the primary replica for a particular RPC. In this case all primary replicas are ordered to abort the transaction. The rest of the work done by Algorithm 2 and 3 are the same as described above. Therefore none of the RPCs is executed in any ways.

Assertion 4: The NTD table will not grow indefinitely and any entry of the table will be eventually deleted.

Proof: An NTD table increases if, and only if, there is a system failure, such as if a replica or a site is down or the network is partitioned into two parts. From our life-cycle assumption for system entities, we know that the maximum time of a system failure is \( T_f \). After that, the replica or the site will rejoin the service and the partitioned network will be re-united. During the recovery, all replicas have to execute Algorithm 4 and the algorithm will delete all NTD table entries at the end of conflict resolution (it is easy to show that Algorithm 5 terminates).

Assertion 4: After the conflict resolution, all outstanding PCs will be either committed or aborted.

Proof: The conflict resolution is carried out by Algorithm 4 after the exchange of NTD tables between replicas of the two parts (failed or alive replicas, or two parts of the partition). The algorithm checks the replica’s NTD table against the received NTD table to see if there are any update conflicts. If no conflict is detected, all PC updates in both NTD tables are committed. If there is a conflict for an update, the algorithm determines the part that has more replicas, or has the smallest sequence number if both parts have the same number of replicas. Then the NTD update from this part is committed, while the NTD update from the other part is aborted. This process continues until all entries of both tables (i.e., all outstanding PCs) are exhausted.

Assertion 6: If a transaction returns PC, the transaction will be notified an OK or an FL return in a finite time.

Proof: The only way that a transaction returns PC is
is free. In this case, our system have a comparable response time compared to a non-replicated system where all real RPCs to all non-replicated servers have to be returned before the transaction can return to the client.

The system response time can be even improved by using the feasibility check results of all primary replicas instead of waiting for the feasibility check result of all replicas of each primary replica. A tentative decision can be made according to these results. An asynchronous procedure is then set up by the transaction manager to process the results returned from all other replicas, and the final transaction result will be decided according to these returns.

7. Examples

In this section we present two examples of the system. The purpose of the first example is to illustrate the algorithms described before, and the purpose of the second example is to show that our system can be used in practical applications.

![Figure 3. An example of RPC transaction processing.](image)

7.1 An Illustrating Example

In this example, we assume that there are two data objects, $d_1$ and $d_2$. Data object $d_1$ is replicated in three replicas $A_1$, $A_2$, and $A_3$, and data object $d_2$ is replicated in $B_1$, $B_2$, and $B_3$. An RPC transaction $T = \{c_1(p_1, d_1), c_2(p_2, d_2)\}$ is to be executed with the help of our RPC transaction manager. Without loss of generality, we also assume that the transaction manager has chosen $A_1$ and $B_1$ as the primary replicas for the execution of the transaction. Fig. 3 shows the RPC transaction processing for this example.

Let us discuss the following three typical scenarios. Other situations can be analyzed in the same manner.

- **No failures.** The RPC transaction manager uses Algorithm 1 to ask primary replica $A_1$ to check the feasibility of $c_1(p_1, d_1)$ and $B_1$ to check feasibility of $c_2(p_2, d_2)$ (the first phase). The two primary replicas then use Algorithm 2 to check the feasibility of the work. Each replica uses Algorithm 3 to do the job. Since there are no failures, all replicas will return OK and the data objects are locked by individual replicas. In the second phase, the two primary replicas are asked to commit the transaction. In turn, these primary replicas ask their replicas to commit the transaction. In that case, all replicas will commit (the second phase of Algorithm 3) the transaction and release the locks. The transaction is then successfully executed. Fig. 4 shows the actions of the RPC transaction manager and the replicas when there is no failures.

![Figure 4. No failures.](image)

- $A_1$ is down and then recovers. Let us first look at the situation that $A_1$ is down. In this case the feasibility check by $B_1$ will return an OK, but the feasibility check by $A_2$ will return a PC. All data objects (except $A_2$'s data object) are locked. In the second phase, the RPC transaction manager will decide to partially commit the transaction and each primary replica will be given the address of an asynchronous handler (but only $A_1$ stores the address). All replicas (except $A_2$) will then execute the RPC and release the locks. Also, replicas $A_1$ and $A_3$ will set the partial commit number for $d_1$ (dpc) to 1 and will record the following information into their NTD tables:

  - $t_{rpc} = \{p_1, d_1\}; t_{pc} = \text{NoOfPCs} = 2; t_{sm} = \text{smallestPC} = 1;
  - t_{data} = d_1; t_{ori} = \text{beforeimage}(d_1); t_{pre} = \text{previous}(d_1); t_{txt} = 0;

  - $B_1, B_2$, and $B_3$ will also have a similar entry added into their NTD tables.

  When $A_2$ recovers and is re-united into the system, $A_2$'s recovery process will cleanup its NTD table and send a re-uniting message and the empty NTD table to all other replicas of $d_1$ (i.e., $A_1$ and $A_3$). Upon receiving the re-unifying message, $A_1$ and $A_3$ will use Algorithm 4 to send their NTD table entries to $A_2$ (the first arrived NTD table will be accepted, others discarded). Then all three replicas will use Algorithm 5 to resolve conflicts based on the same view of the NTD tables. In that case, $A_1$ and $A_3$ will commit the RPC that returned a PC previously by decreasing the dpc number by 1 and delete the entry from their NTD table. Since $A_1$ was the primary replica for this RPC, it also has to use its t.handler field to send an OK message to the RTM. The RTM then can tell the client program that the transaction is upgraded to a commit state. Fig. 5 shows this the actions of this scenario.

If a second transaction $T'' = \{c_2(p_2, d_2), c_2(p_4, d_2)\}$ is submitted to the RTM before $A_2$ is recovered, then all feasibility checks will return PC status and the RTM will decide to do a partial commit. In that case, an entry will be added to all replicas' NTD tables, with the new entry linked to the previous entry, and
the $d_0, pc, f = 1, 2$ fields are increased by 1. Both transactions will be up-graded into the commit state when $A_3$ recovers.

- The network is partitioned into two parts. Suppose that we have the situation depicted in Fig. 6, where the network is partitioned into two parts and that a transaction is submitted to each part of the partition.

![Figure 6. The network is partitioned into two parts.](image)

The feasibility checks of both partitions will return PC states and therefore both transactions will be partially committed by their RTMs. When the two partitions are re-united, both partitions will exchange information of their NTD tables and use Algorithm 5 to resolve conflicts. Since partition 1 has more partially committed replicas, $T$ will be committed in both partitions and $T'$ will be aborted in partition 2.

### 7.2 An Application Example

The system described in this paper has been used in our implementation to improve the reliability of a loosely integrated heterogeneous database system [18]. Fig. 7 depicts the architecture of our loosely integrated heterogeneous database system.

![Figure 7. Architecture of the loosely integrated system.](image)

- Database servers are running as $L_A$ and $L_B$, respectively. A third database server, the Ingres database system, is to be added into the system soon.

- The trader. A trader is a third-party object that links clients and servers in a distributed system [19]. We use a trader to manage the shared information among participating database systems of the loosely integrated database system. If a database system is willing to share part of its information with other database systems, it exports that information as a service offer to the trader. Application programs (clients) wishing to make use of the shared information have to import such service offers from a trader and then access the database(s) concerned. Several types of offers are defined in [18].

- The trading manager. A trading manager is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results.

- Trading agents. Trading agents are appointed by database servers to manage some special service offers. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then a trading agent can be appointed to manage the offer. A trading agent exports the service offer it manages to the trader. The client imports the service offer from the trader and is given the agent's address (and the description of the service, of course). It then calls the agent and obtains the service.

Individual databases in the heterogeneous database system are allowed to maintain their autonomy, yet through the use of traders, they provide a substantial degree of information sharing. The trading manager executes on the same host as the the database server. The trader, trading agents, and user application programs can be executed on any Sun workstations.

The trader is implemented as a server. It provides
export and import operations through remote procedure calls. The trader is a key component of the system — the
system cannot function once the trader fails. Therefore it is replicated. That is, each location has a trading server
running on it. An RPC transaction manager also runs on each location for managing RPC transactions
initialised from the location. These RPC transaction managers have well-known addresses. A fault-tolerant RPC
system [20] is used to implement the reliable communications required by various components of the RPC
transaction management system.

We have carried out the following experiments. Since $L_A$ and $L_B$ are symmetric, the following experiments have
also been carried out for $L_B$, but are not described here. We use $Trader_A$, $Trader_B$, $RTM_A$ and $RTM_B$ to denote
the traders and the RPC transaction managers at locations $L_A$ and $L_B$, respectively.

1. No failures. All read-only operations (imports) to the trader from $L_A$ are directed to the local trader
   $Trader_A$ in $L_A$ by the local RPC transaction manager $RTM_A$. All update operations (exports) to the trader
   are processed as an RPC transaction that has to go through both trader servers. Upon receiving an
   update operation, the RPC transaction manager $RTM_A$ at $L_A$ uses the local trader server $Trader_A$
   as the primary replica and makes the RPC transaction call. When the primary replica returns an OK,
   the update operation is then successful and the user can be sure that both trader servers are to carry the
   update operation.

2. The trader server at $L_A$ ($Trader_A$) fails. All read-only operations to the trader from $L_A$ are directed
   to the remote trader $Trader_B$ in $L_B$ by the local RPC transaction manager $RTM_A$. All update operations
   to the trader from $L_A$ will return a US, then the local RPC transaction manager $RTM_A$ changes the
   primary replica to the trader server $Trader_B$ in $L_B$ and re-submits the RPC transaction. This time a
   PC is returned and the user can be sure that the system is going to perform the update operation on the
   un-accessible replica $Trader_A$ in $L_A$ when it becomes accessible.

3. The RPC transaction manager at $L_A$ ($RTM_A$) fails. In that case the user simply submits all operations
   to the remote RPC transaction manager $RTM_B$ located in $L_B$ (since the addresses of all RPC transaction
   managers are well-known).

4. The trader server $Trader_A$ and the RPC transaction manager $RTM_A$ at $L_A$ fail. All RPC transaction
   requests are directed to the RPC transaction manager $RTM_B$ at $L_B$. An update operation to the trader in
   this case returns a PC and the user can be sure that the system is going to perform the update operation on the
   un-accessible replica $Trader_A$ in $L_A$ when it becomes accessible.

5. The trader server $Trader_B$ at $L_B$ and the RPC transaction manager $RTM_B$ at $L_B$ fail. In that case the
   user submits all operations to the remote RPC transaction manager $RTM_A$ located in $L_A$. An update
   operation to the trader in this case returns a PC and the user can be sure that the system is going to per-
   form the update operation on the un-accessible replica $Trader_B$ in $L_B$ when it becomes accessible.

8. Remarks

A system for building reliable computing over an RPC system is described in this paper. The system combines
the replication and transaction techniques and embeds these techniques into the RPC system. The paper describes
the models for replicas, RPCs, transactions, and the algorithms for managing transactions, replicas, and resolving
conflicts during system recovery. Finally, an informal correctness analysis is carried out and an illustration example
and an application example are described.

References

[1] A. J. Wellings & A. Burns, Programming replicated systems in
CS: A tool for writing fault tolerant distributed software, Di-
gest of Papers:16th Annual Int. Symp. on Fault-Tolerant
Computing, Tokyo, 1988, 56-61.
support for writing fault-tolerant distributed software, IEEE
[4] B. van Reenen & K. Birman, Fault-tolerant programming us-
ing process groups, in F. M. T. Brackey & D. Johnson, eds.,
Distributed Open Systems (Los Alamitos, CA: IEEE Computer
[5] B. Liskov, Distributed programming in ARGUS, Comm. of the
Achieving efficiency and availability in distributed
systems, IEEE Trans. on Software Engineering, SE(1), 1-10,
[7] W. Zhou & B. Mollarni, A system for managing remote pro-
cedure call transactions, J. of Systems and Software, 25(2),
133-149, August 1996.
[8] OSP, OFD distributed computing environment release (Open
Morgan Kaufmann, 1993).
[10] A. Leff & C. Fu, A classification of transaction processing sys-
cedure call transactions, Lecture notes in computer science, 197
(Berlin: Springer-Verlag, May 1991), 572-581.
Biographies

Wanzle Zhou is currently a Senior Lecturer in the School of Computing and Mathematics, Deakin University, Geelong, Australia. He received the B.Eng and M.Eng degrees from Harbin Institute of Technology, Harbin, China in 1982 and 1984, respectively, and his Ph.D. degree from The Australian National University, Canberra, Australia, in 1991. Before joining Deakin University, Dr. Zhou has been a programmer in Apollo/BP at Massachusetts, USA, a Lecturer in National University of Singapore, Singapore, and a Lecturer in Monash University, Melbourne, Australia. His research interests include distributed computing, computer networks, performance evaluation, and fault-tolerant computing. He supervises postgraduate students who carry out research leading toward Ph.D. and M.Sc. degrees. Dr. Zhou is a member of the IEEE and IEEE Computer Society.

Andrzej M. Gocinski received the M.Sc. degree in automatic control in 1968, the Ph.D. degree in control engineering and computer science in 1973, and D.Sc. degree in computer science in 1976 from the Silesian University of Mining and Metallurgy, Krakow, Poland. In January 1992, he took up a Personal Chair in Computing at Deakin University. At the beginning of 1993 he became Head of the School. Professor Gocinski is actively involved in research and teaching. Professor Gocinski's current research activities are in parallel and distributed processing on clusters of workstations, distributed operating systems, applications of distributed systems, networks, and communication protocols. He has been leading research projects (i) to develop a theory of distributed operating systems; (ii) to study design and implement a high performance microkernel-based distributed operating system (RHODOS) to study design issues of distributed operating systems; (iii) to study parallel execution environment and parallelism management in applications executing on clusters of workstations. He has published extensively in refereed journals and conference proceedings, and presented papers at conferences. He supervises postgraduate students who carry out research leading toward Ph.D. and M.Sc. degrees. He is a member of ACM and IEEE Computer Society.
Using Traders for Loosely Integrating Heterogeneous Database Systems

Wanlei Zhou, Philip Hepper and Xidong Wang
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: wanlei@deakin.edu.au

Abstract
This paper presents the design and prototype implementation of a loosely integrated heterogeneous database system. The main goal of the design is to let local databases maintain full autonomy over their databases, yet when willing, they can share some portion of their information. These local databases register with a trader the portion of information that can be shared by using service offers. A service offer can be in the form of data, operations on data, or both. A trading agent is appointed by database systems to managing the services to be provided. The trading agent will be involved in special functions such as schema translation and shared data and/or operations on multiple database systems. The prototype implementation of the loosely integrated heterogeneous database system (involving two different database systems) is now running on a network of Sun workstations.

Key Words: Heterogeneous databases, Traders in open distributed processing, Distributed systems, Client / server model.

1 Introduction
Two approaches are commonly used in integrating heterogeneous database systems. The first is called the unified schema approach [12], and the second the multidatabase approach [13]. Both approaches acknowledge that there are a number of database systems in existence, and the design task involves integrating them into one virtual database system. The unified schema approach starts from individual local databases and translates each participating local conceptual database schema into a common intermediate database schema (a canonical representation). It then integrates each intermediate schema into a global conceptual schema [3]. Sometimes the local external schemas are considered for integrating rather than local conceptual schemas, since it may not be desirable to integrate the entire local conceptual schema in the integrated database.

However, the above approach essentially returns to centralisation by re-integrating the decentralised data into a "composite database." A global/virtual schema is used to describe the information in the databases being composed. Database access and manipulation operations are then mediated through this new conceptual schema. This type of integration can be called tight integration.

The process of integrating existing databases forces control over the local database structures (both conceptually and physically) to be ceded to some central authority. The users of the existing databases may have expended considerable resources in developing their databases and may be reluctant to lose control of them.

The multidatabase approach has no single integrated schema. The shared data is represented either as the actual local conceptual or external schema definition.

In many applications, local databases are not willing to change their own structure or give up control over their data, yet they are willing to share certain information with other database systems. Also, centralising all the local databases into a global schema may be too expensive or even not necessary. The following two application examples:

- A hospital may wish to share a subset of the information about its operating theatre waiting list to some external agent. It is unlikely it would wish to expose other confidential information.

- Two companies are developing a product that involves the use of parts of their own database information. It would be undesirable for either company to divulge rest of their respective databases.

The solution to the above problems is to let individual database system have the full control over their own databases, yet let them decide what portion of the database is going to be shared [13]. No centralised global schema is enforced. We call this type of integration loose integration.

The key issues behind loose database integration are local database autonomy and information sharing [7]. There has been a lot of research on integrating distributed heterogeneous database systems, such as Multidatabase [14], Mermaid [15], InterBase [4], Database [5], Remote-Exchange [10], Pegasus [1], and DIRECT [11]. However, all the existing approaches use passive information sharing. That is, they let some authority (e.g., the global schema) decide what is the content and format of information shared. In this paper we present an integration approach that allows the existing database systems to maintain their autonomy, yet through their willingness, provides a substantial degree of information sharing. The main dif-
ference between our proposal and existing approaches is the modes of information sharing. Our approach uses active information sharing. That is, we let the participating database systems decide the content and format of information to be shared.

2 The trader

2.1 Description of the trader

A trader is a third-party object that links clients and servers in a distributed system [9]. By using a trader, servers can advertise (export) their service offers and clients can get (import) information about one or more exported service offers that match some objectives [6] [2]. Traders have been a subject of international standardisation for some time. The best known and ongoing standardisation work is the ISO Open Distributed Processing project [9].

We use a trader to manage the shared information among participating databases. If a database system is willing to share part of its information with other database systems, it exports that willingness as a service offer to the trader. The following three types of service offers are defined:

- **Data.** Each database system has a collection of data that might be of interest to other database systems. This information can be exported to the trader as a service offer that can then be accessed by others. This type of service offers is designed for direct data sharing.

- **Operation.** A database system may not wish to share its data directly with other database systems. In this case, an operation can be created and exported to the trader for indirect data sharing.

- **Object.** An object contains a piece of data and the operations that manipulate the information. It is essentially the combination of the two previous types of service offers.

Any local database willing to share its information has to export relevant service offers to the trader. Application programs (clients) wishing to make use of the shared information have to import such service offers from the trader and then access the database(s) concerned.

There are two forms of trading:

- **Direct trading.** The database systems export their service offers directly to the trader. The clients import these service offers from the trader and access the relevant database systems directly.

- **Indirect trading.** The database systems appoint a trading agent to manage the service offers. The trading agent is then responsible for exporting the service offer to the trader. After importing the service offer, the clients call the trading agent and all accesses to the database systems have to go through the trading agent.

Figure 1 depicts the direct trading process involving one database system only. In this figure, the database system (DB) exports its service offer to the trader; the client imports the service offer from the trader and the trader returns the offer to the client. The returned offer contains information such as the description of the service and the address of the database system that provides the service; then the client calls the database system directly and the result is returned.

![Figure 1: Trading process: direct trading](image)

Direct trading may not be appropriate in many cases. For example, if some schema translation is needed, or if a service offer involves accessing multiple database systems, then the indirect trading may be used. Figure 2 depicts the indirect trading process that involves two database systems. In this figure, both database systems (DB1 and DB2) are willing to share their information in one service offer. They then appoint an agent and the agent exports the service offer to the trader. The client imports the service offer from the trader and is given the agent's address (and the description of the service, of course). It then calls the agent and obtains the service.

![Figure 2: Trading process: indirect trading](image)

The following questions must be properly answered in order to design such a trader systems:

- **Trading operations.** How do database systems, the trader, and trading agents interact with each other? How do users interact with the trader, trading agents, and database systems?

- **Trading context management.** All services must be registered by the trader before they can be shared. The total set of service offers managed by a trader is called the trading context of the trader. The question is, how are these service offers stored, accessed, and managed?
• An agent is used to provide services that need special treatment, such as a shared service involving two or more database systems. How do these database systems appoint the agent for building a common service for them?

• Is it feasible to build such a trader?

These questions will be addressed in the following sections.

2.2 Service offer data structures

A service offer is actually a set of capabilities that are provided by a server and are to be used by clients. We have defined three types of service offers, namely, data, operations, and objects. In order to describe these service offers in more detail, the following data structures are defined:

• Tables. Tables are the basic unit used in the trader to describe shared information. A database system willing to share its data with other systems must use the following data structure to describe it:

```c
typedef struct
{
    char * sName;  /* attribute name */
    char * sType;  /* attribute type */
    int aLen;     /* maximum length */
} AnAttr;

typedef struct TableStruct {
    char * tableName; /* table’s name */
    AnAttr attr[MAXATTR]; /* attributes */
    int degree;     /* number of attributes */
    int cardinality; /* number of rows */
} TableStruct;
```

The TableStruct data structure defines the name, attributes, type of each attribute, and the size of the table.

• Operations. Another basic unit for describing shared information is an operation (procedure). An operation is defined as follows:

```c
typedef struct
{
    char * procName;  /* procedure name */
    int noOfParameters; /*# of parameters*/
    char ** params;  /* parameters */
    char ** paramTypes; /* types of params*/
} ProcStruct;
```

The ProcStruct data structure defines the name and the parameters (names, types, and number of parameters) of an operation.

• Shared data. A database system can offer a few tables for information sharing. The data structure for such data is as follows:

```c
typedef struct DataStruct {
    char * dataName;  /* name of data */
    int noOfTables;  /* # of tables */
    TableStruct TB[MAXTBRS]; /* tables */
} DataStruct;
```

• Shared operations. Shared operations are offered by database systems that want to share their data indirectly. The data structure for such operations is as follows:

```c
typedef struct
{
    char * transName;  /* name of transaction */
    int noOfProc;     /* # of procedures */
    ProcStruct PR[MAXPRS]; /* procedures */
} TransStruct;
```

• An offer. An offer can be the form of shared data (a set of tables), shared operations (a set of operations), or a shared object (the combination of shared data and operations). The data structure of an offer is as follows:

```c
typedef struct AnOffer {
    char * offerName;  /* offer name */
    int offerType;     /* offer type */
    char * path;       /* context path */
    char * description; /* offer desribe */
    char * addr;       /* service address */
    DataStruct dName;  /* shared data */
    TransStruct tName; /* operation */
} AnOffer;
```

If the offerType = TABTYPE, then the offer is to share a set of tables. The dName will contain the detail of these tables, and the tName will be empty. Similarly, if offerType = TRATYPE, then the offer is to share a set of operations. The tName will contain the detail of these operations and the dName will be empty. If offerType = OBJTYPE, then the offer is to share a set of tables and a set of operations on these tables. In that case, dName will contain the detail of shared tables and the tName will contain the detail of the operations.

2.3 Converting to/from service offer data structures

An exporter wishing to export its services has to use the service offer data structure described in Section 2.2 for describing its offers. If the local database system does not use the service offer data structure in its local definition, a translation is then needed. This can be done through the appointment of an agent. The agent is then responsible for converting the shared portion of the local schema into the service offer data structure. It is also responsible for converting the calls of the shared data structure into the local schema.

An importer imports from the trader an offer described in the service offer structure. It then uses this data structure to access the shared information. If the data structure of the service offer is different with
the importer's data structure, a translation is again needed. We assume that the importer will provide
the mechanism for translating the service offer's data structure into its local data structure.

3 The trading operations

Three sets of operations have been designed and can be used by exporters, importers, and the trader. One common feature of all these operations is the errors returned from the operations. We have defined a global data structure called opErrors that contains most of the possible errors from an operation.

Each operation will return one of the following values when the operation terminates:

#define OP_OK 0
#define OP_ER 1

The global data structure opErrors contains the error descriptions when the operation returns an OP_ER.

3.1 Exporter operations

The basic operation needed by an exporter is to export its service offers to the trader. The exporter may also need to withdraw a service offer if it does not want to provide the service anymore. If some conditions have been changed, the exporter may also want to change the service offer accordingly. It is also necessary for the exporter to obtain some information about an existing service offer placed in the trading context by itself or by other exporters. The following operations have been designed and can be used by an exporter:

- Export. This operation is used by an exporter to export its service offer to the trader. The trader has a well-known address and is hidden inside the operation. This operation has the following format:

  int export(offer, createOpt)
  AnOffer offer;
  int createOpt;

  Before calling this operation, the exporter has to fill in the offer structure with proper information, such as the type of the offer, the description of the offer, and the details of the offer. Most importantly, it has to give the preferred context path name for the offer to be stored within the trading context. When the offer reaches the trader, the trader will use this path name to store the offer into the trading context. The structure of the trading context is discussed in Section 4. The exporter also needs to specify its address in the offer structure.

- Withdraw. This operation is used by an exporter to withdraw an offer that it placed at the trader at some prior time. The operation has the following format:

  int withdraw(offerName, path)
  char *offerName;
  char *path;

  Both the offerName and path must be the same as used by the exporter in the export operation. The path is used by the trader to find the service offer in the trading context and the offerName is used to confirm the name of the offer to be deleted from the trading context.

- Replace. This operation is used by an exporter to replace an offer that it placed at the trader previously. The operation has the following format:

  int replace(prevPath, prevName, offer)
  char *prevPath;
  char *prevName;
  AnOffer offer;

  The prevPath and prevName contain the context path and the offer name that the exporter used during the export operation for placing the particular offer. This operation can change all the details of the service offer such as the name, the description, the data and/or operations, and so on. Even the context path name can be changed. In that case, the old service offer will be deleted from the trading context and then a new service offer will be stored by using the new context path and the new offer name contained in the offer data structure.

- Describe. This operation is used by an exporter to get some information about a particular offer. The operation has the following format:

  int describe(path, offerName, offerType, description);
  char *path;
  char *offerName;
  int *offerType;
  char *description;

  The path is the context path name of the service offer that the exporter wants to know about. The operation then returns the type of the service offer and the descriptions about the service.

3.2 Importer operations

The basic operation needed by an importer is to import a service offer from the trader. In many cases, it is also necessary for an importer to browse through the trading context and to obtain some descriptions about a particular service offer. The following operations have been designed and can be used by an importer:

- Import. This operation lets the importer import a service offer from the trader. The importer has to be aware of the context path and the offer's name of the service offer before it can invoke the import operation. The operation has the following format:

  int import(path, offerName, offer)
  char *path;
  char *offerName;
  AnOffer *offer;
After receives the import request, the trader will search its context space for the given path and the offer's name. If the search is successful, the selected service offer will be stored in the offer and returned to the importer.

- Describe. This operation is the same as the describe operation for an exporter.

- List. The describe operation only returns the descriptions of a particular service offer. Sometimes it is necessary to know about a set of offers that provide the same services. This operation is used to return the types, context path names, and descriptions of a set of service offers. The operation has the following format:

```c
int list(offerName, noOfOffers, offerTypes, paths, descriptions)
```

The operation takes the offerName as the input parameter and asks the trader to find out all service offers with the same offerName. If the operation is successful, the noOfOffers contains the number of service offers that match the offerName and the offerTypes, paths and descriptions contain the types, context path names, and descriptions of each of these service offers.

3.3 Management operations
All the management operations are used by the trader to manage the trading context. These operations allow the trader to add in new service offers, to delete or modify existing offers, to find the description about a specific offer, and to browse through the trading context. These operations can be (and actually have been) used by a utility tool that talks to the trader directly for managing the trading context. We will not go into details of these operations since the functions of these operations are similar to those of the operations provided to the exporters and importers. The only difference is that these operations are now performed locally within the trader, or between the trader and the management utility.

4 Trading context management
The trading context of the trader consists of all the service offers registered with the trader. Associated with each service offer is the information that describes the service. That may include, for example, the name/address of the agent or database system that offers such service, the type of the offer, and a brief description of the service (e.g., a brief introduction of how to use the service).

Service offers of the trading context are managed by the trader as a tree structure, similar to the UNIX directory structure. Figure 3 describes an example of the trading context.

When exporting a service offer, the agent or the database system should specify the intended path of the trading context and the offer's name for the service offer. The trader then uses the path and the offer's name to record and locate the service offer within the trading context.

The trading context also contains a linked list for offerName fields. This linked list is used for obtaining descriptions and other information about a group of offers that have the same offerName.

The actual trading context uses only one set of context path names and two sets of pointers. One set of the pointers is used for constructing the context tree structure and the other set of pointers is used for constructing the linked list of offerNames.

The client can browse through the trading context and then find the desired service offer (through the list operation, for instance). The client can also obtain a more detailed description from the database or it appointed agent.

5 The trading agents
A trading agent integrates services involving two or more database systems, or provides some special management for the service offer. If two databases are heterogeneous, translators for transforming local schemas into a virtual schema may be necessary. The virtual schema is determined by the agent and may be negotiated by the participating database systems. Although we only use a label “Apoint” in Figure 2 to represent the process of trading agent appointment, the process would actually involves several communication steps among the participating database systems and the trading agent.

In our prototype implementation, we have not implemented a standard facility for trading agent appointment and for negotiations among participating database systems. At this moment trading agents are hand-written programs (with the help of a precompiler, see Section 6 for details) that perform the functions such as accepting local offers from participating database systems, format translating between local offers and the virtual offer, exporting the virtual offer to the trader, and processing requests from clients. The local offers of participating database systems are decided before the program of a trading agent is written.
6 Prototype implementation

Currently the trader has been implemented on a set of networked Sun workstations. Two database systems are used in the prototype: an Oracle database system and a Mini SQL database engine [8]. An Ingres database system and a POET object-oriented database system have been planned to be added into the system soon.

6.1 Architecture

Figure 4 depicts the architecture of our loosely integrated heterogeneous database system. Currently two database systems (Oracle and Mini SQL) are running on two separate Sun workstations. The trader, trading agents, and user application programs can be executed on any Sun workstations.

![Architecture Diagram]

Figure 4: Architecture of the loosely integrated system

A trading manager is built on every participating database system for performing all common tasks of trading preparation and management. It is responsible for such tasks as checking the validity of trading requests, forming local offers, executing the service, and returning request results. By using the trading manager, we can reduce the duplicating part of each trading agent and concentrate on the work of the trading agent such as schema translation and combination. Of course, it also adds one more level of interprocess communication.

One may argue that having one trading agent for each offer is a waste of resources. It is true to some extent, but it is also the mechanism that provides diversity of service requests and loose integration. These agents are application-oriented. For instance, if a special application requires some information from two or more heterogeneous database systems, the trading agent provides an easy mechanism for loose integration. A service offer can be deleted from the trading context if it is no longer needed.

The rapid prototyping tool described in [16] is used in the prototype implementation.

The trader is implemented as a server. It provides export and import operations through remote procedure calls (RPCs). The trader also uses a well-known port for accepting calls from clients. The context space is currently located in the main memory because of the small size. It should be implemented on a file system if the space required becomes large enough.

The trading agent acts as both a server and a client. To the user program, a trading agent is a server because it provides services that the user program has imported from the trader. But to a trader program, a trading agent is a client because it exports service offers to the trader. The interprocess communication between trading agents and the trading manager is through sockets.

6.2 Offer definition files

We use an offer definition file (ODF) to define an offer and then through a precompiler called offer frame generator, the source files of the offer defined in the ODF will be generated. Currently only C source code is generated. Listing 1 shows the syntax of an offer definition file.

Listing 1: Offer definition file syntax

```
ODF ::= BEGIN
   INC := [ INC ]
   OFFER ::= [ TABLES ]
   PROC ::= END
   INC ::= [ INC ]
   INC ::= Include: filename ;
   OFFER ::= Offer Name: variable;
            Offer Path: string;
            Offer Description: variable;
            Offer Address: HOST, PORT;
   HOST ::= string
   PORT ::= integer
   TABLES ::= Table: string from [variable];
            TBS
   TBS ::= TB { TB }
   TB ::= Table Name: string ; ATTR
   ATTR ::= { ATTR }
   ATTR ::= Attr: declarator ;
   PROC ::= Procedures: string; GPS ;
            End Procedures ;
   GPS ::= OP { OP }
   OP ::= Name: string ; PARMS
   PARMS ::= ( PARMS )
   PARMS ::= Param: CLASS: declarator ;
   CLASS ::= in | out | inout
```

Most of the descriptions of Listing 1 are self-explanatory. We use a modified BNF to denote the syntax of definition files, where [x] means that x can appear 1 to many times and {x} means that x can appear 0 to many times. The "variable", "integer", "string", and "declarator" have the same meanings as in the C programming language. Comments are allowed in the definition file. They are defined the same as in the C programming language (using /* and */). The semantics of an ODF file will be made clear in Section 5.4.

6.3 Offer frame generator

After a programmer sends an offer definition file to the offer frame generator, the generator first does syntax checking. If no errors are found, several program source files are generated. These generated files can be used by programs that use the service offer, such as the trading agent, the trading manager, and the user program. A `makefile` is also generated for testing the
service offer by using the generated driver programs. That is, when using the make utility, the executable files for the trading agent, the trading manager, and the user program will be generated. Figure 5 shows the input and output of the offer frame generator. By default, source code for the trading agent is always generated. If the user does not want trading agent to be generated, the source code for trading manager and user application program will be a little different (for instance, the trading manager in this case will include operations previously located in the trading agent).

Figure 5: Input and output of the offer frame generator

6.4 Example applications

We use a simple example to show the application of the tools and the idea of loosely integrating heterogeneous database systems. Suppose we have a database about off-campus student records and it is stored in an Oracle database. The table definition is assumed to be as follows (the database may contain other tables):

OFFCAMPUS(sid integer, surname char[20],
street char[20], suburb char[20],
city char[10], country char[12],
phone char[16], major, ...)  

Assume that we have decided to share those student records that have major = 'Computing' in some applications. We may want to define an offer as follows:

/* Offer from off-campus student database */
BEGIN
 Include: offCampus.sql
 Name: ofcs;
 Offer Path: /University/Faculty/Computing;
Offer Description:
"A table for off-campus students majoring in Computing. In Oracle database, an operation to change student address";
Offer Address: fredo.deakin.edu.au, 6500;

Tables: 
 StudentTable;
 Name: Student from OFFCAMPUS;
 Attrib: int sid;
 Attrib: char surname[20];
 Attrib: char street[20];
 Attrib: char suburb[20];
 Attrib: char city[10];
 Attrib: char country[12];

Attrib: char phone[16];
 End Tables;

Procedures:
 Name: changeAddress;
 Param: in: integer sid;
 Param: in: char street[20];
 Param: in: char suburb[20];
 Param: in: char city[10];
 Param: in: char country[12];
 End Procedures;

The offer has one table which shows the details of students that are majoring in "Computing", and an operation that changes a student's address. The trading agent is to be executed on machine fredo.deakin.edu.au and is to use socket port 6500. The offer is going to be stored in the trading context path /University/Faculty/Computing under the name of ofcs. The table of the offer is created from a database definition file named offCampus.sql, which contains a CREATE TABLE statement for the OFFCAMPUS table. After we send this file to the offer frame generator, the following files will be generated:

ofcs.h  Header file, must be included by trading agent, user program, and trading manager.
ofcsTA.c  Trading agent driver file
ofcsTAStub.c  Trading agent stub file
ofcsTA0ps.c  Framework of trading agent
ofcsTM.c  Trading manager driver file
ofcsTMStub.c  Trading manager stub file
ofcsTM0ps.c  Framework of trading manager
ofcsAP.c  User application driver file
ofcsAPStub.c  User application stub file
makefile  All these files use the data structure defined in Section 2.2. After using the make utility, three executables will be generated:
ofcsTA  Trading agent program
ofcsTM  Trading manager program
ofcsAP  User application program

The ofcsTM0ps.c file contains such operations as security checking, view creating, offer creating, offer executing, and so on. The ofcsTA0ps.c file contains such operations as offer exporting, format translating, and so on. Note that some of the operations (such as the changeAddress operation defined in the definition file) are not fully defined in these files. Instead, only frameworks (dummy procedures) are defined for such operations. Their details are to be programmed by the programmer.

Now suppose we have another ISAM database containing on-campus student records and is defined as follows:

ONCAMPUS(sid integer, studentName char[30],
 address char[40], phone char[10],
major, ...)

Assume that both databases want to share their student records with “Computing” major in some applications. We can define the following offer to accommodate the differences of the two databases:
The offer contains only one table. It is going to be executed on the baragund.deakin.edu.au machine and is to use socket port 5700. The table is created from two database definition files named offCampus.sql and onCampus.sql. These two files contain the definitions for OFFCAMPUS and ONCAMPUS tables, respectively. The offer is going to be stored in the trading context path /University/Faculty/Computing under the name of Offer.

The trading agent in this case will be responsible of operations including forming the virtual table from the two tables, directing user requests to different database trading managers, accepting results from trading managers, and converting the results into the virtual table format. Note that because the addresse attribute in the global table combines a few attributes from the student table of the off-campus database, any update to this field will then be prohibited by the trading agent.

7 Remarks

The design and prototype implementation of a loosely integrated heterogeneous database system is described in this paper. The main contribution of this paper is the introduction of the active participation of information sharing by the database systems and the introduction of data and operation sharing. It also shows that building such a system that loosely integrates heterogeneous database systems is possible.

References


A Hybrid Protocol for Managing Replicated Data

Wanlei Zhou and Robert Holmes
School of Computing and Mathematics
Deakin University
Blackburn Rd., Clayton, VIC 3168, Australia
Email: wanlei@deakin.edu.au

Abstract

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a distributed computing system. Keeping the replicated data consistent and available during the presence of failures can however become rather difficult. This paper presents the design of a hybrid replica control protocol that attempts to maximise availability and minimise communication overhead, by combining the advantages of two common replica control protocols into one. The protocol was simulated using SimJava, a process-based discrete event simulation package. The results from the simulations showed that not only did the hybrid algorithm maintain a high level of availability, it did so while minimising communication overheads.

1 Introduction

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a distributed computing system [5]. However, although considerable research effort have been directed towards the design of replication-control protocols, replication is still viewed as a "necessary evil" [8]. Most existing replication-control protocols are either un-efficient or too complicated to be implemented [1, 2]. Reference [9] gives a comprehensive overview of replication techniques and annotated bibliographies of selected literature on replication techniques and example systems.

Primary copy algorithms are based on the static ownership of data [1], where one copy of a data object is designated as the primary copy. The node that maintains the primary copy of the data object is therefore responsible for providing mutual exclusive access to the data object. Any other node that maintains a non-primary copy, known as a slave copy, wishing to perform an update must request the primary copy to perform the update request on their behalf. Once the primary copy is updated, it will be propagated out to all the nodes that maintain slave copies. The protocol is easy to implement. However it has the limitation that if a node that maintains a primary copy fails, or the link to the primary copy fails, then an update operation can no longer be performed until the node or the link becomes available again.

A number of replica control protocols have been suggested based on the dynamic ownership of data. One such protocol is the Majority Consensus Voting algorithm [7]. The algorithm was based on the constraint that in order to perform an update operation, the requesting node must obtain permission from the majority of all nodes in the distributed database system. Having no central control over the data makes the algorithm more robust compared to the primary copy approach. However, the communication cost of such a protocol is much higher than that of the primary copy approach. Also, during a network partitioning, if none of the partitions contain the majority of nodes, then no update operations can proceed in any partition.

This paper aims to develop a replica control protocol that is communicationally efficient, and highly available. The protocol must maintain these design goals during abnormal system behaviour such as node and communication failures. Analysis of the protocol will be conducted, and compared to the primary copy and majority voting algorithms.

2 Protocol Design

The algorithm design is based on the primary copy approach and the majority voting approach. The operation of the algorithm can be broken down into two main phases: the normal system operation and the abnormal system operation. During normal system behaviour, a simple primary copy approach is employed. Each node in the system knows the location of the primary copy. When a node receives an update request
from a client, it passes it onto the primary copy where the update proceeds and will be replicated to all other nodes.

During abnormal system behaviour, a hybrid primary/majority algorithm is employed. In the event of a communication failure resulting in a network partition, only the nodes in the same partition as the primary node can perform update operations. The ideal situation would be to have the primary node in the partition that contains a majority of nodes. This may or may not occur naturally without intervention from the algorithm. If it does, a majority of the nodes can perform updates, maximising availability, while still minimising communication overhead. If it doesn’t, the system becomes less available, which is not what we desire. Therefore, a new primary node must be elected in the partition that contains a majority of nodes, in order to maintain maximum availability.

The current design of the algorithm assumes that if communication with the primary node is unavailable, then the primary node is in another network partition. In normal system operation, communication with the primary node may be unavailable directly, but indirect communication through another node may be possible. For simplicity, the algorithm only tries to communicate directly with the primary node. If communication fails, it does not try to communicate indirectly through another server. The algorithm design could be easily altered to accommodate such a situation.

It is also assumed that all the clients and their parent servers will be in the same network partition after a communication failure. This means there are no timeout mechanisms incorporated into the design for requests sent from the clients. By assuming they are in the same partition, a client will always receive a response.

Figure 1 outlines the hybrid protocol. The detailed explanation of the protocol can be found in [4].

3 Simulation and Analysis

3.1 Simulation Design

The simulations were written using a process-based discrete event simulation package developed in Java by the Department of Computer Science at the University of Edinburgh, called SimJava (http://www.dcs.ed.ac.uk/~home/bass/simjava/simjava-1.0/). Since the protocol hopes to achieve higher levels of availability and lower communication costs, the simulation does not concern itself with time. Instead, when simulating the protocol, we are more interested in the number of messages required to either pass a transaction (communication cost) and also the number of passed or failed transactions (availability) with and without network partitioning.

The simulation of the hybrid protocol consists of four main classes, rcphybrid(server), controlmodule(server) and client(). The rcp.hybrid() class is the main class for starting off the simulation and setting up the environment. The client() and server() classes simulate the behaviour of clients and servers in a normal distributed system, while the controlmodule() simulates failures in the distributed system. For the purpose of simplicity, each of the simulations are only concerned with replication of one data object. When a client requests an update operation, it is based on this data object.

Implementation of the primary copy simulation basically followed much of the design philosophy of the hybrid protocol, except primary re-election was not enabled. The majority voting simulation also followed the same design principle as described above. Statistical collection methods for the majority voting algorithm did require some slight modifications due to the number of parallel requests that a server can handle without trying to acquire locks.
3.2 Testing Methodology

The main testing process involved running each of the simulated protocols under a number of different simulation scenarios. In each of the scenarios, all three replica control protocols contained the same simulation environment variables which included communication delay, acknowledgement timeout delay, update processing delay and reply processing delay. Such environment variables can have a significant effect on the outcome of a simulation, therefore during the testing procedure, it was imperative that such values remained the same. Testing the many different combinations of these environment variables and scenarios would have been an extremely large task, of which they may or may not have provided results which would be useful to draw conclusions from. Therefore, each protocol was tested using the same environment variables with varying system sizes and loads. The system size ranged from 3, 5 and 10 servers, each of which were tested with different loads of 1, 5, 10 and 25 requests per server. In each scenario, the three protocols were simulated fifty times, with each of the different system sizes and loads.

The following testing scenarios were concerned:

1. No failure: The first scenario was concerned with testing each of the protocols under normal system behaviour. No network partitioning occurred, and all servers could communicate.

2. Random failure: The second scenario involved introducing a network failure causing the network to partition. The failure was random such that it could occur at any point during the simulation. It may happen that it occurs at the start, middle, or end of the simulation. Such a scenario tries to simulate a real world example, in which update requests may already be in process when the network failure occurs.

3. Fixed failure: The last testing scenario also involved introducing a network failure. However, instead of randomly generating the failure at some point in time during the simulation, the final scenario generates a network failure at the beginning of the simulation. This scenario attempts to more fairly illustrate how well each protocol functions during a network failure.

3.3 Availability Analysis

Availability is the probability that a system is usable during a certain time interval. We will assume that the availability of the replicated data object is the probability that it is in a state permitting access [6]. To estimate the general availability of each protocol, we will look at the number of update requests that have been accepted during the simulations. The higher the number of passed requests, the more available the system must have been in order to pass those requests.

![Figure 2: Passed requests during random network failures](image)

Figure 2 shows the mean number of passed requests for the fifty simulation runs, with 3, 5 and 10 servers during random network failures. In each of the three system sizes, the hybrid protocol has performed well, each passing on average 81% of all update requests. The majority voting and primary copy algorithms both struggled to achieve poor pass rates of only 46.4% and 50.3% respectively. This was to be expected of the primary copy approach since it is extremely vulnerable to network failures. There is a 50% chance for the primary node to be in either partition during a network failure, which then may or may not contain a majority of nodes. This greatly affects the number of requests that can be passed. The majority voting algorithm should perform much better on average than the primary copy approach. This has failed to be the case in all three system sizes. This is because of the deadlock detection and resolution algorithm used in our simulation where many update requests were forced to fail in order to allow a local node to perform an update request.

Contributing to such a poor result is the fact that the algorithm does not try to re-acquire a lock on another node once a lock has been canceled. It simply fails the entire update request. As shown in Figure 2, the more the system size and load increases, the chance of a possible deadlock increases and the worse the majority voting algorithm performs.

Table 1 shows a summary of pass rate percentages along with the average pass rate for each of the protocols. Reference [4] contains a detailed collection of the statistical data. Using analysis of variance tech-
niques (ANOVA), we can determine if the protocol type has a significant effect on the pass rate of update requests. The null hypothesis that we will test is $H_0: \mu_{\text{primary}} = \mu_{\text{majority}} = \mu_{\text{hybrid}}$. That is the true pass rate mean is the same for each protocol type tested. In other words, the protocol type does not have a significant effect on the pass rate. The alternate to the null hypothesis is

$$H_a: \text{Not all true protocol pass rate means are the same.}$$

The results of the ANOVA test follows.

$H_0: \mu_{\text{primary}} = \mu_{\text{majority}} = \mu_{\text{hybrid}}$

$H_a: \text{Not all true protocol pass rate means are the same}$

$\alpha = 0.05$

<table>
<thead>
<tr>
<th>Source</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor</td>
<td>8618.28</td>
<td>2.00</td>
<td>4309.14</td>
</tr>
<tr>
<td>Error</td>
<td>1412.77</td>
<td>33.00</td>
<td>42.81</td>
</tr>
<tr>
<td>Total</td>
<td>10031.05</td>
<td>35.00</td>
<td>286.60</td>
</tr>
</tbody>
</table>

Table 2: Test results

$F^* = 100.65$

From these results, we reject $H_0$: because the value $F^*$ fell within the critical region of the $F$ distribution $F(2, 33, 0.05)$. It can therefore be concluded that the protocol type does have a significant effect on the pass rate of update requests.

The random failure scenario attempts to simulate a real world network failure by allowing them to occur at any point in the simulation. This provides a good view of how the system may run in a normal environment, but does not particularly show how well each actually performs during the network failure itself. Each of the protocols was therefore tested with a fixed network failure, occurring at the beginning of each simulation.

Figure 3 shows the mean number of passed requests during fixed network failures at the start of each simulation. The hybrid protocol again has performed better than the other two algorithms, however this time only achieving a pass rate of approximately 64% (Table 3). The 17% decrease in pass rate from random failures was considered the amount of requests that were passed during a time of no failure the random simulation scenario. The majority value algorithm again has performed extremely poorly to the deadlock detection problems described at achieving only a 36.9% pass rate. The 9.8% decrease can also be considered the amount of passed requests that were passed during a time of no failures in previous simulation scenario. One will also notice the standard deviation for the majority voting algorithms during random and fixed failures, are significantly higher than the standard deviations for the primary and hybrid protocols. This suggests that the primary mean pass rate for the majority voting algorithm does not reflect the population mean pass rate as the hybrid and primary algorithm means do.

Again, using ANOVA we can determine if the protocol type has a significant effect on the pass rate of update requests during a network failure.

$H_0: \mu_{\text{primary}} = \mu_{\text{majority}} = \mu_{\text{hybrid}}$

$H_a: \text{Not all true protocol pass rate means are same}$

$\alpha = 0.05$

$F^* = 70.80$

The second ANOVA test also rejected the null hypothesis, because the value $F^*$ fell within the critical region of the $F$ distribution $F(2, 33, 0.05)$. The protocol type therefore does have a significant effect on pass rate of update requests during network failure.
Table 3: Passed requests during fixed network failures (%)

<table>
<thead>
<tr>
<th>Requests per server</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>25</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total requests</td>
<td>5</td>
<td>15</td>
<td>30</td>
<td>75</td>
</tr>
<tr>
<td>Passed Primary (%)</td>
<td>46.3</td>
<td>56.5</td>
<td>56.3</td>
<td>46.3</td>
</tr>
<tr>
<td>Passed Majority (%)</td>
<td>77.9</td>
<td>77.9</td>
<td>77.9</td>
<td>77.9</td>
</tr>
<tr>
<td>Passed Hybrid (%)</td>
<td>66.7</td>
<td>66.7</td>
<td>66.7</td>
<td>66.7</td>
</tr>
</tbody>
</table>

Table 4: Test results

<table>
<thead>
<tr>
<th>Source</th>
<th>SS</th>
<th>df</th>
<th>MS</th>
</tr>
</thead>
<tbody>
<tr>
<td>Factor</td>
<td>4397.28</td>
<td>2.00</td>
<td>2198.64</td>
</tr>
<tr>
<td>Error</td>
<td>2198.64</td>
<td>33.00</td>
<td>65.05</td>
</tr>
<tr>
<td>Total</td>
<td>5422.03</td>
<td>35.00</td>
<td></td>
</tr>
</tbody>
</table>

3.4 Communication Overhead Analysis

To measure the communication costs, we simply record the number of messages sent on average to perform an update request. This does not include messages sent as part of the replication of a committed update request, since this will be the same for any protocol type used.

Figure 4: Messages per pass request with no failures

Figure 4 shows the mean number of messages sent for a pass update request. Here we can see that both the primary and hybrid protocols performed identically during no network failures, with a mean number of message sent of 3.58. As explained in the design of the hybrid algorithm, a simple primary copy approach is used until a network failure occurs, which explains why they both have the same communication costs. The majority voting algorithm as expected is more message intensive. From Figure 4, we can see that the primary and hybrid protocols both maintain low message counts even as the system size is increased. However, the message count of the majority voting algorithm increases dramatically as the number of servers increase.

Figure 5: Messages per pass request with random failures

Figure 5 shows the mean messages sent per pass requests during random failures. From the results, it is clear that the primary and hybrid protocols have a significantly lower communication overhead than the majority voting algorithm.

3.5 Overall Performance Analysis

Figure 6 shows both the mean percentage of requests passed and the mean number of messages for a pass request for all simulation scenarios. It provides an overall picture of the operation of the hybrid protocol in respect to its design goals.

Figure 6: Overall percentage of passed requests / pass messages per request

From figure 6 we can see that the hybrid protocol has outperformed both the primary and majority voting al-
algorithms in the percentage of overall requests passed. In particular, it has performed significantly better during random network failures. While maintaining this higher level of availability, the average number of messages required to pass an update request has been minimized. The communication overhead is only slightly higher than that of the primary copy approach.

The results produced were very favourable for the hybrid protocol. It was shown that it maintained a higher level pass rate than either of the other protocols. 81% of requests were passed during random network failures, and 64% were passed during the fixed network failure, both of which were significantly higher. These higher pass rates were also achieved while still maintaining a low communication overhead of approximately 3.58 messages per pass request. The poor performance of the majority voting algorithm was discovered not to be because of the algorithm itself, rather the implementation we used in relation to deadlock detection and resolution. Therefore, it is not justifiable to draw conclusions from the availability analysis of the majority voting algorithm. This problem does not affect the communication overhead analysis results of the majority voting algorithm.

4 Conclusion

In this paper, a hybrid data replication algorithm was presented that combined the advantages of two common replica control protocols, majority voting and the primary copy, into one. Its design goals were to maximise availability of a data object, while still maintaining low communication overhead. During the state of normal system operation, the algorithm would function exactly the same as the primary copy approach, meeting the design goal of providing low communication overhead, and good response times. In the event of a network failure, the hybrid algorithm would provide maximum availability by re-electing the primary node in the partition that contained a majority of nodes. Meeting the design goal of maximising availability. The results of the simulations confirmed the success of the algorithm in meeting its design goals.

Possible directions of further work could include implementing the protocol in a real working environment to see if it can produce the results comparable to the simulation environment.

References

[1] S. Ceri, M. A. W. Houstema, A. M. Keller, and P. Samarat. A classification of up...
The Design and Simulation of a Hybrid Replication Control Protocol

Wanlei Zhou and Robert Holmes
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
Email: {wanlei, rholmes}@deakin.edu.au

Abstract. Replica control protocols in distributed database systems are responsible for the management and maintenance of replicated (redundant) data. Maintaining replicated data improves performance and increases availability. Keeping the replicated data consistent and available during the presence of failures can however become quite difficult. This paper presents the design of a hybrid replica control protocol that attempts to maximise availability and minimise communication overhead, by combining the advantages of two common replica control protocols into one. The protocol was simulated using SimJava, a process-based discrete event simulation package. The results from the simulations showed that not only did the hybrid algorithm maintain a high level of availability, it did so while minimising communication overheads.

Keywords: Distributed computing, replication control protocols, fault tolerance, simulation.

1 Introduction

The management of replicated data in a distributed system involves maintaining the integrity and availability of data at each remote location. This is achieved with the use of a replica control protocol [5, 11]. As companies move toward systems that are more open and distributed, replication of data is becoming increasingly important in the ability to provide data that is current, correct and available, which is a key factor in maintaining a competitive advantage over rivals.

Replica control protocols must synchronize concurrent operations performed on the same data object by transactions at different locations. It must also provide this synchronization during the event of system failures. Although considerable research effort has been directed towards the design of replication-control protocols, most existing replication-control protocols are either unefficient or too complicated to be implemented [2, 3]. Reference [6] gives a comprehensive overview of replication techniques and annotated bibliographies of selected literature on replication techniques and example systems.

The simplest replica control protocols are based around static ownership of data [2]. Primary copy algorithms designate one copy of a data object as the primary copy. The node that maintains the primary copy of the data object is therefore responsible for providing mutual exclusive access to the data object. Any other node that maintains a non-primary copy, known as a slave copy, wishing to perform an update must request the owner of the data (the primary copy node) to perform the update request on their behalf. Once the primary copy is updated, it will be propagated out to all the nodes that maintain slave copies. The benefit of such a protocol is that it is easy to implement compared to other known protocols, such as the Multidimensional Voting [1] protocol. It does however have the limitation that if a node that maintains a primary copy fails, or the link to the primary copy fails, then an update operation can no longer be performed until the node or the link becomes available again.

A number of replica control protocols have been suggested based on the dynamic ownership of data. One such protocol originally suggested by Robert Thomas [10] introduced the concept of voting. Called the Majority Consensus Voting algorithm, it was based on the constraint that in order to perform an update operation, the requesting node must obtain permission from the majority of all nodes in the distributed database system. Having no central control over the data makes the algorithm more robust compared to the primary copy approach. However, the communication cost of such a protocol is much higher than that of the primary copy approach. It also has an undesirable characteristic when communication failures partition the network. That is, if none of the partitions contain the majority of nodes, then no update operation can proceed in any partition. A replica control protocol that provides mutual exclusive access and improved availability during network partitioning is required.

This paper aims to develop a replica control protocol that is communicationally efficient, and highly available. The protocol must maintain the design goals during abnormal system behaviour such as node and communication failures. Analysis of the protocol will be conducted, and compared to the primary copy and majority voting algorithms.
2 Protocol Design

The algorithm design is based on the primary copy approach and the majority voting approach. The main idea behind the proposed hybrid algorithm is that during normal system operation, the algorithm follows a normal primary copy approach. In the event of a communication failure causing a network partition, it degrades to a hybrid primary/majority algorithm, therefore minimising communication overhead as much as possible, while still maintaining a high level of availability. The detailed explanation of the protocol can be found in [7].

A number of difficulties arise in the design of the algorithm. The first one concerns the re-election of the primary node. If a network partition occurs, how can we guarantee that there is only ever one primary node in the system at any one time, if communication among nodes in different partitions is unavailable? The answer is we can’t [8]. We can however prevent a primary node from performing update operations if it is not in the majority partition, based on the majority voting algorithm.

The second problem concerns pending update requests at the primary node, and what we do with them if a network partition occurs. If a primary node has update requests queued, and a network partition occurs, the following four scenarios could occur.

1. The current primary node is in the majority partition, and the requests are from clients/servers in the same partition.
2. The current primary node is in the majority partition, but the requests are from clients/servers in a different partition.
3. The current primary node is not in the majority partition, and the requests are from clients/servers in the same partition.
4. The current primary node is not in the majority partition, but the requests are from clients/servers in the majority partition.

For scenario one, the answer is simple. The primary node continues updating and notifies the requesting entity of the update completion. Scenario three is also easily dealt with. Simply deny all update requests and notify the requesting entity. Scenarios two and four however are much more complicated. Communication between the primary node and the requesting entity is unavailable. In the case of scenario two, should the updates proceed, or should they be denied on the basis that a response to the requesting entity can not be returned. If the requests are allowed, how can the requesting entity be notified. In scenario four, the update requests must obviously be denied, but again, how do the requesting entities know. As far as they are concerned, the request could have been updated before the failure, but the acknowledgement was lost.

The current design of the algorithm assumes that if communication with the primary node is unavailable, then the primary node is in another network partition. In normal system operation, communication with the primary node may be unavailable directly, but indirect communication through another node may be possible. For simplicity, the algorithm only tries to communicate directly with the primary node. If communication fails, it does not try to communicate indirectly through another server. The algorithm design could however be easily altered to accommodate such a situation.

It is also assumed that all the clients and their parent servers will be in the same network partition after a communication failure. This means there are no timeout mechanisms incorporated into the design for requests sent from the clients. By assuming they are in the same partition, a client will always receive a response.

3 Simulation

The simulations were written using a process-based discrete event simulation package developed in Java by the Department of Computer Science at the University of Edinburgh, called SimJava (http://www.dcs.ed.ac.uk/home/hase/simjava/simjava-1.0/). Since the protocol hopes to achieve higher levels of availability and lower communication costs, the simulation does not concern itself with time. Instead, when simulating the protocol, we are more interested in the number of messages required to either pass or fail a transaction (communication cost) and also the number of passed or failed transactions (availability) with and without network partitioning.

3.1 The Hybrid Protocol Simulation

The simulation consists of four main classes, rcp_hybrid(), control.module(), server() and client(). The rcp_hybrid() class is the main class for starting off the simulation and setting up the environment. The client() and server() classes simulate the behaviour of clients and servers in a normal distributed system, while the control.module() simulates failures in the distributed system. For the purpose of simplicity, each of the simulations are only concerned with replication of one data object. When a client requests an update operation, it is based on this data object.

The rcp_hybrid() class is responsible for setting up the simulation environment and initialising the simulation entities such as servers and clients. The class also contains many simulation environment variables such as number of servers, number of requests per server, communication delay, update processing delay, reply processing delay, acknowledge-ment timeout, and network failure time.

By altering these simulation environment variables, the protocols behaviour can be analysed and
evaluated in different scenarios.

The main role of the client entities is to simply send update requests to its server at randomly generated time intervals. Update requests are sent as an event to the server with a delay specified by the communication delay environment variable. Once a request has been sent, the client entity waits for a response, of which it will either be confirmation that the update completed or failed. Each client maintains information on the total number of requests sent, and total passed and failed requests. When a response arrives, the appropriate statistical information is altered.

The client then waits for a randomly generated time interval before sending the next update request. Once the total number of requests that a client is required to send have been sent, the client entity gathers all its statistical information, and sends it to the control module for system wide statistical calculations.

A server entity is responsible for handling update requests based on the design of the hybrid algorithm. Message passing is simulated by sending events to other server entities with a simulated communication delay. The server entity is also responsible for maintaining statistical information about the current update request, such as how many messages (events) have been sent in order to achieve mutual exclusion. Each server entity maintains a server information table containing current connections and communication status of other servers in the distributed system. This information is modified on the receipt of events from the simulation control module. When an event is received from the simulation control module, it must be processed with a higher priority than other events that the server receives as part of the hybrid algorithm. This is to ensure that when a simulated network partitioning occurs, it happens before any other algorithm processing continues. As in a real system, when a failure occurs, it interrupts the current processing of the algorithm.

The control module is responsible for simulating network failures and partitioning in the distributed system. When a communication failure is supposed to occur causing the network to partition, the control module creates a failure object that consists of an array of integers. This array of integers represents which nodes are in which network partition. Each index of the array represents the different server id’s, and the contents of the array, either 0 or 1, represents what partition the server is part of. Once the failure object has been created, it is sent with an event to each server in the system. From the failure object, a server can determine what partition it is in, and what other servers it can communicate with during the failure.

3.2 Primary Copy and Majority Voting Simulations

Implementation of the primary copy simulation basically followed much of the design philosophy of the hybrid protocol, except primary re-election was not enabled. The majority voting simulation also followed the same design principle as described above. Statistics collection methods for the majority voting algorithm did however require some slight modifications due to the number of parallel requests that a server can send out when trying to acquire locks. Since the majority voting algorithm requires allocation of distributed resources, deadlock detection and resolution mechanisms were also required. This can become an extremely complex problem in distributed systems as discussed in [4]. Therefore, the implementation of deadlock detection and resolution in the majority voting simulation is very basic. It allows nodes to request resources and wait for them without restriction. As nodes receive lock requests, they are placed on a resource required queue. When the resource (logical lock) becomes available, the node requesting the resource at the top of the queue is granted access. As nodes receive update requests from clients, they do not lock their local resource until they acquire all remote resources. Nodes try to acquire locks from nodes with the lowest priority id’s first. Once this is achieved, the local node will then send a request to the node that holds the local lock, and ask it to release it. It can then proceed with the update and release its remote locks. This basic method in conjunction with the order in which nodes try to acquire locks, resolves many deadlock situations, but by no means all.

4 Testing and Analysis

4.1 Testing Methodology

The main testing process involved running each of the simulated protocols under the following different simulation scenarios:

1. Scenario One - No failure: The first scenario was concerned with testing each of the protocols under normal system behaviour.

2. Scenario Two - Random failure: The second scenario involved introducing a network failure causing the network to partition. The failure was random such that it could occur at any point during the simulation.

3. Scenario Three - Fixed failure: The last testing scenario also involved introducing a network failure. However, instead of randomly generating the failure at some point in time during the simulation, the final scenario generates a network failure at the beginning of the simulation. This scenario attempts to more fairly
illustrate how well each protocol functions during a network failure.

Each protocol was tested using the same environment variables with varying system sizes and loads. The system size ranged from 3, 5 and 10 servers, each of which were tested with different loads of 1, 5, 10 and 25 requests per server. In each scenario, the three protocols were simulated fifty times, with each of the different system sizes and loads.

4.2 Availability Analysis

We will assume that the availability of the replicated data object is the probability that it is in a state permitting access [9]. To estimate the general availability of each protocol, we will look at the number of update requests that have been accepted during the simulations. The higher the number of passed requests, the more available the system must have been in order to pass those requests.

<table>
<thead>
<tr>
<th>Protocol</th>
<th>Requests per server</th>
</tr>
</thead>
<tbody>
<tr>
<td>Primary</td>
<td>1</td>
</tr>
<tr>
<td>Hybrid</td>
<td>2</td>
</tr>
<tr>
<td>Majority</td>
<td>3</td>
</tr>
<tr>
<td>Total</td>
<td>6</td>
</tr>
</tbody>
</table>

Figure 1: Legend for graphs in section 4.2

Figure 2: Passed requests during random network failures

Figure 2 shows the mean number of passed requests for the fifty simulation runs, with 3, 5 and 10 servers during random network failures. In each of the three system sizes, the hybrid protocol has performed well, each passing on average 81% of all update requests. The majority voting and primary copy algorithms however both struggled to achieve poor pass rates of only 46.4% and 50.3% respectively. This was to be expected of the primary copy approach since it is extremely vulnerable to network failures. There is a 50% chance for the primary node to be in either partition during a network failure, which they may or may not contain a majority of nodes. This greatly affects the number of requests that can be passed. The majority voting algorithm however should perform much better on average than the primary copy approach. This has failed to be the case in all three system sizes. After much analysis of the design and implementation of the majority voting simulation, we have discovered its disappointing performance is a result of the deadlock detection and resolution algorithm used. Such a basic method, described in section 3.3, results in many update requests being forced to fail in order to allow a local node to perform an update request. During the original design, the method described seemed to resolve many deadlock situations. However, the downside to such an approach results in many update requests being failed, even when a deadlock situation does not exist. Therefore removing its distinct advantage, of providing a higher pass rate than that of the primary copy approach.

Contributing to such a poor result is the fact that the algorithm does not try to re-acquire a lock on another node once a lock has been canceled. It simply fails the entire update request. As shown in Figure 2, the more the system size and load increases, the chance of a possible deadlock increases and the worse the majority voting algorithm performs.

The random failure scenario attempts to simulate a real world network failure by allowing them to occur at any point in the simulation. This provides a good view of how the system may run in a normal environment, but does not particularly show how well each actually performs during the network failure itself. Each of the protocols was therefore tested with a fixed network failure, occurring at the beginning of each simulation.

Figure 3: Passed requests during fixed network failures

Figure 3 shows the mean number of passed requests during fixed network failures at the start of each simulation. The hybrid protocol again has performed better than the other two algorithms, however this time only achieving a pass rate of approximately 64%. The 17% decrease in pass rate from random failures to fixed failures can be considered the amount of passed requests that were passed during a time of no failures in the random simulation scenario. The majority voting algorithm again has performed extremely poorly due to the deadlock detection problems described above, achieving only a 36.9% pass rate. The 9.5% decrease can also be considered the amount of passed requests that were passed during a time of no failures in the previous simulation scenario. One will also notice that the standard deviation for the majority voting algorithms during random and fixed failures, are sig-
significantly higher than the standard deviations for the primary and hybrid protocols. This suggests that the sample mean pass rate for the majority voting algorithm does not reflect the population mean pass rate as well as the hybrid and primary algorithm means do.

4.3 Communication Overhead

The communication overhead of a replica control protocol is the number of messages required to provide mutual exclusion. This overhead is detrimental in two ways. Firstly, the physical cost in dollars if the distributed system is run over a leased network and secondly, higher communication costs usually result in slower response times. It is therefore desirable to minimise the communication overhead of the replica control protocol. To measure the communication costs, we simply record the number of messages sent on average to perform an update request. This does not include messages sent as part of the replication of a committed update request, since this will be the same for any protocol type used.

![Figure 4: Legend for graphs in section 4.3](image)

![Figure 5: Messages per pass request with no failures](image)

Figure 5 shows the mean number of messages sent for a pass update request. Here we can see that both the primary and hybrid protocols performed identically during no network failures, with a mean number of message sent of 3.58. As explained in the design of the hybrid algorithm, a simple primary copy approach is used until a network failure occurs, which explains why they both have the same communication costs. The majority voting algorithm as expected is more message intensive. From Figure 5, we can see that the primary and hybrid protocols both maintain low message counts even as the system size is increased. However, the message count of the majority voting algorithm increases dramatically as the number of servers increase.

Figure 6 shows the mean messages sent per pass requests during random failures. From the results, it is clear that the primary and hybrid protocols have a significantly lower communication overhead than the majority voting algorithm.

4.4 Overall Performance Analysis

As stated in the design of the hybrid algorithm, the goal of the protocol was to minimise communication overhead, while still providing maximum availability. The following graph show both the mean percentage of requests passed and the mean number of messages for a pass request for all simulation scenarios. It provides an overall picture of the operation of the hybrid protocol in respect to its design goals.

![Figure 7: Legend for graphs in section 4.4](image)

![Figure 8: Overall percentage of passed requests / pass messages per request](image)

From figure 8 we can see that the hybrid protocol has outperformed both the primary and majority voting algorithms in the percentage of overall requests passed. In particular, it has performed significantly better during random network failures. While maintaining this higher level of availability, the average number of messages required to pass an update request has been minimised. The communication overhead is only slightly higher than that of the primary copy approach.
The results produced were very favourable for the hybrid protocol. It was shown that it maintained a higher level pass rate than either of the other protocols. 81% of requests were passed during random network failures, and 64% were passed during the fixed network failure, both of which were significantly higher. These higher pass rates were also achieved while still maintaining a low communication overhead of approximately 3.58 messages per pass request. The poor performance of the majority voting algorithm was discovered not to be because of the algorithm itself, rather the implementation we used in relation to deadlock detection and resolution. Therefore, it is not justifiable to draw conclusions from the availability analysis of the majority voting algorithm. This problem however does not affect the communication overhead analysis results of the majority voting algorithm.

5 Conclusion

In this paper, a new hybrid data replication algorithm was presented that combined the advantages of two common replica control protocols, majority voting and the primary copy, into one. It's design goals were to maximise availability of a data object, while still maintaining low communication overhead. During the state of normal system operation, the algorithm would function exactly the same as the primary copy approach, meeting the design goal of providing low communication overhead, and good response times. In the event of a network failure, the hybrid algorithm would provide maximum availability by re-selecting the primary node in the partition that contained a majority of nodes. Meeting the design goal of maximising availability.

The results of the simulations confirmed the success of the algorithm in meeting its design goals. The algorithm achieved a higher success rate of update requests than both the primary copy and majority voting algorithms during normal and abnormal system behaviour. The high success rate was also achieved while still maintaining communication overhead comparable to the primary copy approach. However, due to inadequate implementation of the majority voting simulation, the success of the hybrid protocol is somewhat biased. In theory, the majority voting algorithm should have a success rate of update requests the same as the hybrid approach. This is because during a network failure that causes a network partition, both algorithms are designed to allow a majority of nodes to continue performing update requests. Although there is this minor problem with the majority voting simulation, the overall comparative analysis of the three algorithms is not affected and should still be considered valid. This is simply because the comparison to the majority voting algorithm was not to show that the hybrid protocol had a higher pass rate, but to demonstrate that it provided a high pass rate while still maintaining a low communication overhead. Likewise with the primary copy approach, it was not compared to the hybrid algorithm to show that it had a lower communication overhead, but to demonstrate that it had a higher pass rate during a failure than the primary copy approach.

References


Dealing with Network Partitioning: a Case Study

Wanlei Zhou
School of Computing and Mathematics, Deakin University, Clayton, VIC 3168, Australia. wanlei@deakin.edu.au

Weijia Jia
Department of Computer Science, City University of Hong Kong, Kowloon, Hong Kong. wjia@cs.cityu.edu.hk

Abstract. Network-based applications are vulnerable to network partitioning failures. The challenge is therefore to let the application continue its operations during a network partitioning, yet to reconcile the effects of incompatible operations when the communication is restored. In this paper we present a strategy that allows every part of a distributed application to continue its operations during a network partitioning. When the network partitioning is recovered, a reconciliation process will bring the system to a consistent state.

Key Words: Fault-tolerant computing, Network partitioning, Network-based applications, Replication, Databases.

1 Introduction

Many of today's application systems are network-based. That is, these systems are built on the top of a set of computers, linked through a communication network, or even the Internet. Such systems provide the possibility of sharing information and peripheral resources. Furthermore, these systems can improve the performance of a business application through parallel execution of programs, load balancing and sharing. Network-based applications are also characterised by enhanced availability and increased reliability.

However, network-based applications also generated some serious challenges and problems. One of such challenges is the withstanding of network partitioning failures [5]. A network partitioning failure splits the network into two or more disjoint parts. Processes of a network-based application within the same part can communicate with each other, but they cannot communicate with processes of the application located in other parts. In order for the application to be continuously operational, data and processes must be replicated in the network. However, application processes may perform some incompatible operations that can result in inconsistent data during the network partitioning. The challenge is to let the application continue its operations during a network partitioning, yet to reconcile the effects of incompatible operations when the communication is restored.
In this paper we have developed a strategy that allows every part of a network-based application to continue its operations during a network partitioning. When the network partitioning is recovered, a reconciliation process will bring the system to a consistent state.

This paper is structured as follows. In Section 2, we describe the background and the architecture of our network-based application. In Section 3 we present the replication strategy for normal operations. The replication strategy for dealing with network partitioning and recovery is described in Section 4. In Section 5 we present some performance evaluation results. In Section 6 we conclude the paper.

2 The Application

In this section, we summarise the architecture and basic features of our network-based application. The application is a prototype of a network-based sales system spread across three cities: Melbourne, Sydney and Geelong. Each city has a database which records the inventory of the regional warehouse and a group of salespersons that rely on the database for their sales activities. We call these databases as Melbourne inventory DB, Sydney inventory DB and Geelong inventory DB, respectively. Each city also stores replications of databases of other cities. Therefore, a salesperson can also sale things stored in other cities' inventory databases, although majority of sales will be from the local inventory database. Figure 1 depicts the architecture of the network-based sales system.

![Figure 1. Architecture of the network-based sales system](image)

A transaction management system (consisted of a set of transaction managers) runs between the user (a salesperson) and the databases. Java embedded
Web browser such as Netscape, Internet Explorer, HotJava Browser, allow users to easily fetch any Java programs at run time. We choose one such browser as the interface between user, which is the front end, and our transaction management system which is a set of Java implemented programs. Java applets in a Web page, triggered by a user when viewed by the Web browser, are links to execute certain transaction operations. In the back end, we use Java Database Connectivity (JDBC) for the servers in the transaction management system to access the physical data sources in various locations.

Java and JDBC, from our previous experience for building of a mobile transaction management system involving heterogeneous databases [7], have been proven to be an ideal tool for network computing. Their flexibility of coding once and running everywhere nicely suit our needs for heterogeneous environment. Specifically, when we work on a distributed database system, JDBC frees developers by its goal of being a DBMS independent interface, a uniform interface to different data sources. It actually completes part of the work of schema translation which is the key issue dealing with different database systems. When contacting the actual data source, all one has to do is to address the URL (Unified Resource Location) in the network. The lower level data retrieving and manipulating is handled by JDBC drivers. From the user’s point of view, that means one single query sentence can be used to access data in all kinds of sources, provided the database system being used has installed specific JDBC drivers. Figure 2 shows the information flow in our network-based design for the sales system.

Salespersons which have access to local Web servers use the (Java enabled) Web Browser to locate a Web page designed for our system. There are Java applets embedded in the page containing fields for user input and for displaying feedback. After selecting his/her transaction service, the salesperson sends the request through the Web which actually triggers the execution of server programs on the fly. Transaction managers in Figure 2 receive user requests and execute them on individual databases. These transaction managers are also responsible for the propagation of updates to replicas. JDBC is the bridge between transaction managers, which are all Java programs, and different databases. JDBC
services are called by the server programs embedded with JDBC driver objects.

On the transaction managers' side, we use Java's Remote Method Invocation (RMI) which is a powerful tool for structuring distributed client/server applications. The transaction managers in our system are remote server objects listening for calls from applets all the time. They have the detailed information of the server groups which actually execute individual sub-transactions included in a user's request. When a transaction manager receives a call from the applet, it first decides which server groups to contact. It then acts as a secondary client to invoke those server groups using RMI. After the result is returned to the transaction manager, it passes on to the applet to display to the user and do the necessary job of coordinating among replicas. The server groups are the transaction managers for the primary/non-primary replicas of the same database which are stand alone Java programs sitting on different machines. The JDBC driver objects are embedded in these programs.

3 Replication Strategy: Normal Situations

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a network-based computing system [1, 4]. However, although considerable research effort has been directed towards the design of replication-control protocols, replication is still viewed as a "necessary evil" [6]. Most existing replication-control protocols are either inefficient or too complicated to be implemented [2, 3]. Reference [4] gives a comprehensive overview of replication techniques and annotated bibliographies of selected literature on replication techniques and example systems.

In this section we describe our replication strategy informally. A formal description and the performance evaluation of the replication strategy will be discussed in another paper.

3.1 Transactions

We first define some terms. The Sydney inventory database on the Sydney site is called the primary database (or primary, in short) of Sydney site. Similarly, the Melbourne inventory database is called the primary database of Melbourne site and the Geelong inventory database is called the primary database of Geelong site. Other databases stored on a site are called replica databases (or replicas, in short). If a transaction issued from a site (i.e., a salesperson of that site originated the transaction on behalf of a customer) only accesses its primary database, then we call such a transaction as a primary transaction (PT). A transaction issued from a site accesses data of a replica database is called a replica transaction (RT). Transactions propagated from other sites and to be executed on the current site are called propagated transactions. Naturally, we may have propagated primary transactions (also called propagated forced transactions (PFT) since these transactions will be forced executed on the current site, see Section 3.2) and propagated
replica transactions (PRT). Without loss of generality, we define all transactions to be update-oriented transactions.

We also assign access priority to transactions issued from each site according to the business logic. A transaction issued from Melbourne site has the highest access priority (numbered 0) since the Melbourne site is the location of the business headquarter. Transactions issued from Sydney site have medium access priority (numbered 1) and transactions issued from Geelong site have the lowest access priority (numbered 2).

The execution of a transaction on a primary database can have three results. First, the transaction’s execution can be completely successful. For example, if a transaction requires the sale of 10 HP1100 LaserJet printers and the primary inventory database has 20 in stock, then the transaction can be successfully committed. Second, the transaction’s execution is not successful initially, but can become successful by the generation of a fulfillment transaction [5]. For example, if a transaction requires the sale of 20 HP1100 LaserJet printers and the primary inventory database only has 10 in stock, then a fulfillment transaction will be generated to order 10 (or maybe 15 if the lower limit of the stock is 5) more HP1100 LaserJet printers. In this case, the transaction is also regarded to be successful. The execution of a fulfillment transaction will be carried out before the execution of the original transaction and it follows the same strategy of the execution of a normal transaction. We regard the two successful cases as the same. Last, the transaction cannot be executed successfully and a fulfillment transaction cannot be generated. For example, if a transaction requires the sale of 10 HP1100 LaserJet printers, the primary inventory database has none in stock and the manufacturer has stopped producing the product, then the transaction fails.

The execution of a transaction on a replica database can also have three results. First, the execution can be completely successful (commit), which means that the transaction is executed successfully on the replica database and has been successfully executed on the primary database. Second, the transaction has been successfully executed on the replica database, but the execution result on the primary database is still unknown. We call this transaction as a partially committed transaction. Third, the transaction fails on the replica database. A partially committed transaction will be upgraded into commit if the execution on the primary database is successful, or will be down-graded into fail if the execution on the primary database fails.

3.2 Normal operations

We discuss the operations of a transaction manager on a site when there is no failure. Users connected to each site can request two types of transactions, the primary transactions and the replica transactions, to the transaction manager. Also, the transaction manager can receive two types of transactions, the propagated forced transactions and the propagated replica transactions, from other transaction managers. In addition to these transactions, the transaction manager can also receive returned results about the execution on propagated transactions.
For each site, upon originated a primary transaction, it processes the primary transaction right away and returns the result back to the user. If the primary transaction has been successfully executed locally, then the transaction manager on the site propagates the primary transaction to other two sites. The transaction managers on the other two sites will force these propagated primary transactions (called propagated forced transactions) to be executed on their replicas.

Upon originated a replica transaction, the transaction manager on a site processes the replica transaction using its primary database (if the transaction also accesses the primary database) and the replica databases. If the processing is successful, the transaction is regarded as partially committed and the user is notified of the tentative execution result. If the partially committed transaction accesses one replica, the transaction is sent to the primary site of the replica as a propagated replica transaction for further processing. A forward mark is attached to this transaction noting that the third replica should execute this transaction as a propagated forced transaction if the primary site's execution is successful. If the partially committed transaction accesses two replicas, it is sent to the primary site with a higher access priority. A forward mark is attached to the transaction to note that the primary site with a lower access priority should execute the transaction as a propagated replica transaction.

Upon receiving a propagated forced transaction, the transaction manager on a site executes the transaction on the relevant replica database(s). This type of transactions have a higher priority and will be executed immediately (i.e., before any replica transactions and any propagated replica transactions).

Upon receiving a propagated replica transaction, the transaction manager on a site executes the transaction. If the execution of the transaction is successful and the transaction's forward marked is for this site, then an OK result is returned to the sender of this transaction. If the execution is successful and the transaction's forward marked is for the third site, then the transaction will be sent to the third site as a propagated forced transaction or as a propagated replica transaction, depending on the nature of the forward mark. If the execution fails, then all effect of the transaction will be cleared from this site and a failed result will be returned to the sender of this transaction.

Upon receiving a returned result on a propagated replica transaction, the transaction manager on a site will analyse the result. If it is an OK and this site originated the transaction, then the transaction (which was partially committed) will finally commit (i.e., the effect of the transaction will be made permanent and the user is notified of the result). If this site did not originate the transaction, then the effect of the transaction will be made permanent on this site and the OK result will be sent to the origin of the transaction (the origin can be identified easily since there are only three sites in our system). If the result is failed and this site originated the transaction, then all effect of the partially committed transaction will be cleared from this site and a failed result will be returned to the user. If this site did not originate the transaction, then all effect of the transaction will be cleared from this site and a failed result will be sent to the origin of the transaction.
Fig. 3. Transaction processing. PT: Primary transaction. PFT: Propagated forced transaction. Si, Sj, Sk: Sites. PFT.dm: The destination mark of the PFT. RT: Replica transaction. Ri, Rj, RK: Replicated databases on Si, Sj, and Sk, respectively. Fi: Primary database on Si. PRT: Propagated replica transaction. PRT.fm: The forward mark of the PRT. Sj.prio: Access priority of transactions issued from site Sj.

The successful execution of a transaction on a primary database may generate a new fulfillment transaction, in that case, the effect of the fulfillment transaction will be enclosed into the original transaction. For a primary transaction and a replica transaction (if it also accesses the primary database), this can be done easily since they have not involved other sites yet. For a propagated forced transaction, there is no such issue. The processing of a propagated replica transaction involves more work when fulfillment transactions are generated. Let us examine an extreme case. For example, if a replica transaction originated from Sydney site has been successfully executed on both Melbourne and Geelong replicas, then it will be sent to the Melbourne site first (since it has a higher access priority) with a forward mark noting that it has to be sent to the Geelong site as a propagated replica transaction. Now assume that when the propagated replica transaction is executed on the Melbourne site, it generates a fulfillment transaction. Then this fulfillment transaction will be sent back to Sydney site as a propagated forced transaction. The effect of the fulfillment transaction will also be incorporated into the original transaction before it is forwarded to the Geelong site. If the execution of the transaction on Geelong site also generates a
fulfillment transaction, then the fulfillment transaction will be sent to both Melbourne and Sydney sites as a propagated forced transaction. The generation, incorporation, and propagation of fulfillment transactions are an integrated part of our transaction management system.

4 Replication Strategy: Network Partitioning and Recovery

4.1 Operations during network partitioning

When one site is partitioned from other two sites, then it cannot propagate its transactions to other sites, nor it is able to receive any transactions from other sites. If a site originates a primary transaction and the execution of the transaction is successful, then the primary transaction is logged into a stable storage (a hard disk file) together with a local timestamp (all logged transactions require a local timestamp to keep the local ordering). A destination mark is also attached to the transaction noting that it has to be executed on the two replicas. The transaction is considered committed even the system has to propagate the logged primary transaction to the two replicas when the network partitioning is recovered.

If this site originates a replica transaction and the execution is successful, then the replica transaction is logged into the stable storage. The transaction
is considered partially committed. The final fate of the transaction can only be decided when the network partitioning is recovered and the logged replica transaction is executed by the relevant primary site(s). The destination mark of the transaction can be decided using the same strategy for processing replica transactions (as described in Section 3.2). Figure 3 shows the processing of transactions.

When two sites are partitioned from the third site, then the two sites can communicate with each other but they cannot communicate with the third site. If a site originates a primary transaction and the execution of the transaction is successful, then the primary transaction will be propagated to the communicating site for a forced execution. The destination mark of the transaction is set to the third site and the transaction is logged into the stable storage. If a site originates a replica transaction and the execution is successful, then the transaction is considered partially committed. If the transaction can be sent to the communicating site according to the strategy described in Section 3.2, then it is sent to the communicating site with a correct forward mark for further propagation (either as a propagated forced transaction or as a propagated replica transaction). Otherwise the transaction (with a correct forward mark) is logged into the stable storage and the destination mark of the transaction is set to the disconnected site.

If a site receives a propagated forced transaction, the transaction will be executed the same way as if the network partitioning were not present.

If a site receives a propagated replica transaction, the transaction manager on that site will execute the transaction. If the execution is successful, then the transaction will be logged into the stable storage, together with the original forward mark and a destination mark for the disconnected site. If the execution fails, then all effect of the transaction will be cleared from this site and a failed result will be returned to the sender of this transaction. The processing of transactions in the above cases is also depicted in Figure 3.

If a site receives a returned failed result on a propagated replica transaction, it knows that the communicating site failed to execute the transaction. It then clears all effect of the partially committed transaction from this site and a failed result will be returned to the user. Figure 4 also depicts this situation.

4.2 Operations during recovery

When a network partitioning recovers, all sites will go into the recovering process where a set of recovery operations will be executed before any new transactions can be executed. The recovery process has the following three stages:

- Propagation stage. At this stage all sites group all their logged transactions into two sets according to the destination mark of each transaction. Each set will be sent to the relevant site as a single log message. If a site does not have any logged transactions for a particular site, then a log message of an empty set will be sent to the latter site instead.
Ordering stage. When a site has received log messages from other two sites, it first orders all propagated forced transactions according to their access priority. Transactions with the same access priority are ordered according to their local timestamps. Then all propagated replica transactions will be ordered the same way as the propagated forced transactions. i.e., firstly the access priority then the local timestamps.

Execution stage. After the ordering of all transactions, the transaction manager on a site will execute these transactions according to their ordering. Firstly the propagated forced transactions are executed. The execution results of these transactions must be OK since these transactions have been successfully executed on their primary sites. Then the propagated replica transactions are executed according to the ordering. The strategy for the execution of the propagated replica transactions during the execution stage is very similar to the strategy described in Section 3.2. For example, if the execution of a propagated replica transaction is OK and the forward mark of the transaction is for the current site, then the transaction commits and an OK result is returned to the sender of this transaction. Upon receiving the OK result, the sender of this transaction will commit the transaction. If the sender is also originated the transaction, it will notify the user (using an asynchronous mechanism) that the partially committed transaction has been upgraded to fully commit.

The execution of propagated replica transactions may also generate fulfillment transactions. These fulfillment transactions will be processed using the same strategy as described in Section 3.2.

All databases stored on each site will be in a consistent state when these sites complete the recovery process.

5 Performance Issues

We have conducted our preliminary evaluation within a lightly loaded local subnet (10M Ethernet) of Sun Sparc workstations using the JDK 1.1.2 version. The reason to use a lightly loaded subnet instead of the Internet to carry out the test is that our main purpose of the test is to determine the overhead of our replication strategies. To restrict the test into a local subnet allows us to filter out the unpredictable delays that may be introduced in an Internet environment.

We tested five types of transactions:

1. Primary transaction accessing the primary only ($T_p$).
2. Replica transaction accessing one replica only ($T_r$).
3. Replica transaction accessing the primary and one replica ($T_{p+r}$).
4. Replica transaction accessing the two replicas only ($T_{r+r}$).
5. Replica transaction accessing the primary and all replicas ($T_{p+r+r}$).

Each transaction type was tested 20 times under two scenarios: no network partitioning failure or with a network partitioning failure. During the testing, we
measured the time from the point that a user sends out a request till the time a response is received by the user. All transactions were designed to be able to execute successfully and therefore no fulfillment transactions were generated.

Table 1 lists our test results. The table shows the average response time for each case. The standard deviations for all the cases are 1% to 5%. To increase the readability of the table, we omitted the standard deviation of all the results.

<table>
<thead>
<tr>
<th>Transaction Type</th>
<th>Partial Commit (ms)</th>
<th>Final Commit (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>No NP</td>
<td>With NP</td>
</tr>
<tr>
<td>( T_P )</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>( T_r )</td>
<td>217</td>
<td>215</td>
</tr>
<tr>
<td>( T_{p+r} )</td>
<td>219</td>
<td>218</td>
</tr>
<tr>
<td>( T_{r+r} )</td>
<td>217</td>
<td>219</td>
</tr>
<tr>
<td>( T_{p+r+r} )</td>
<td>220</td>
<td>219</td>
</tr>
</tbody>
</table>

Table 1. Performance evaluation (NP - Network Partitioning failure, \( T_{NP} \): Time duration of the network partitioning failure)

Table 1 shows that, for a primary transaction \( T_P \), a network partitioning failure does not affect the response time (considering up to 5% standard deviations, the two times are the same). This is because that the commit result for \( T_P \) is returned immediately after the execution of the transaction, no matter if a network partitioning failure occurs or not.

For a replica transaction using only one replica (\( T_r \)), a network partitioning failure does not affect the time for returning a partial commit. However, a network partitioning failure greatly affects the time for returning the final result of the transaction. In addition to the time duration of the network partitioning \( T_{NP} \), the response time also increased by about 37ms. This time is the overhead during the recovery process, mainly due to the transaction managers exchanging their transaction logs. Note also that there is a long delay (about 200ms) between a partial commit till a final commit. This is caused by the propagation to and execution of \( T_r \) by the primary site of the replica.

Similarly, a network partitioning failure does not affect the time for returning a partial commit for other transaction types (\( T_{p+r}, T_{r+r} \) and \( T_{p+r+r} \)). Also, the overhead of the recovery process is about 37ms in all cases. It is also interesting to note that the times used to return a partial commit result are the same for all transaction types (excluding \( T_P \), of course). This is because in our prototype implementation, a single table is used to store information for all primary and replicas. It can be anticipated that if different tables were used to store different inventory databases, then these times would have some differences.

Another observation is that transactions \( T_{r+r} \) and \( T_{p+r+r} \) require longer time to reach the final commit decision. This is because in both cases, the final decision needs to be made by all three sites before the decision is returned.
6 Conclusions

In this paper we have presented a case study for surviving network partitioning failures through the use of a network-based application. Our major contributions are: (a) the introduction of a partial commit state for transactions on replicated databases, (b) the introduction of a fulfillment transaction as a part of normal database operations, and (c) the combination of partially commit states and fulfillment transactions in dealing with network partitioning failures.

The concept of partially committed transactions was introduced in [8] in dealing with remote procedure call transactions. The extension of this concept into replicated database transactions allows a database transaction to be proceeded even in a network partitioning. The concept of fulfillment transactions was introduced in the Totem project [5] to deal with network partitioning. The extension of this concept as a part of normal database operations not only allows fulfillment transactions to be used to reduce the effect of inopera
tible operations during a network partitioning, but also captures the essence of business transactions.

References

The Design and Evaluation of A Token-Based Independent Update Protocol

Wanlei Zhou

School of Computing and Mathematics Deakin University Clayton, VIC 3168, Australia, wanlei@deakin.edu.au

Weijia Jia

Department of Computer Science City University of Hong Kong Kowloon, Hong Kong, wjia@cs.cityu.edu.hk

Abstract

This paper presents the design and evaluation of a token-based protocol supporting independent updates in replicated objects. The paper makes three major contributions. Firstly, a token is used to simplify the control of update propagation and ordering. Secondly, a partial commit state is introduced to support independent updates and enhance the efficiency of transaction processing. Thirdly, a data partition methodology based on priorities is adopted to reduce update conflicts. We evaluate our protocol in comparison of two existing replication-control protocols in terms of transaction processing efficiency.

Key Words: Distributed objects, Transaction processing, Replication, Fault tolerance, Performance evaluation.

1 Introduction

Many operations in a replicated object system are organised as transactions [1]. In classical research literature, transactions are characterised to have the properties of atomicity, consistency, isolation, and durability (ACID). The two-phase-commit (2PC) protocol is the most widely used protocol in commercial databases to maintain the ACID properties of transactions. Such a protocol has some intrinsic disadvantages, such as cost, delay, and reduced availability [2]. Many applications, such as flight reservation system, inventory control system, computer supportive cooperative system, etc., do not really require such restricted properties [3, 4]. Based on the classical transaction model, many novel transaction models have been proposed in recent research literature. Most of them are motivated by workflow and design applications, where long-lived activities need to be structured into semi-independent atomic steps [5, 6]. However, the efficiency and ease of implementation are still the main obstacles to the widely use of replication techniques.

The purpose of this paper is to propose an approach for supporting efficient independent updates and maintaining transaction atomicity. Our proposed protocol is called the token-based independent update (TBIU) protocol. We arrange all sites containing replicated objects as a logical ring and let a token circulate among these sites. Each site can execute transactions on a subset of their replicated objects locally, resulting partially committed transactions. Transactions accessing data objects outside the subset have to be queued up until the site has the token. Only the site holding the token is allowed to propagate transactions to all sites.

The main advantages of this scheme are (1) efficient: it does not use distributed 2PL protocol, only local 2PC protocol is used; (2) no global rollbacks: when there are conflict operations, only the partially committed transactions may be rolled back locally; (3) easy to implement: the proposed scheme is built on top of an existing efficient and reliable atomic multicast protocol [7].

2 System Models

2.1 Transaction and Partial Commit

A replicated object system provides a set of (n) services: S = {S1, S2, · · · , Sn}. Each service Si is implemented through a logical object R = {P0, D0}, where D0 = {d01, d02, · · · , d0n} is the set of data managed by R and P0 = {p01, p02, · · · , p0n} is the set of methods provided by R to manipulate D0.

We use R = {R1, R2, · · · , Rm} to denote the logical objects and D = {D1, D2, · · · , Dm} to denote the data items of R.

Each logical object R = {Ro, R1, · · · , Rk} is supported by a set of (k) replicated physical objects (replicas). Each replicated physical object Rj also has two parts, Rj = {Pj, Dj}, where Dj = {d1j, d2j, · · · , dnj} is the set of data managed by Rj and Pj = {p1j, p2j, · · · , npj} is the set of methods provided by Rj for manipulating Dj.

Let P = UP, and D = UD, for i = 1, 2, · · · , n and j = 1, 2, · · · , m, then a method call to the replicated objects can be expressed as c(p, d), p ∈ P, d ∈ D. That is, the call uses method p to manipulate data item d.

We define a transaction as

T = {c1(p1, d1), c2(p2, d2), · · · , cn(pn, dn)},

where c(i, pi, di) is a method call, and pi ∈ P, di ∈ D. We
express the set of all methods and the set of all data items of
T as $T.P = \{p_1, p_2, \ldots, p_l\}$ and $T.D = \{d_1, d_2, \ldots, d_k\}$,
respectively. The semantics of a transaction is that after is-
suing the transaction, the state of the transaction will be in
one of the following:

- OK: there is no error occurs and all $c_i$ of $T$ have been
  executed on all sites,
- FL: if any one of the $c_i$ fails, all executed method calls
  of $T$ will be rolled back, and
- PC: the transaction is successfully executed on the loca-
  l site, waiting to be propagated to other sites.

In addition to the two normal states (commit and abort),
we define a partial commit (PC) state. A call $c_i(p_i, d_i)$ re-
turns PC if and only if it has been performed locally within
one replica and will be propagated to other replicas later.
When the transaction is propagated to other replicas, the
client which issued the transaction will be notified of the
final result (OK or FL).

2.2 Access Priority and Data Partitioning

In applications such as flight reservation systems, inventory
control systems, and computer cooperative work systems,
different transactions may have different priorities to access
various data items. We use such priorities to partition data
stored on various sites and associate the priorities to trans-
actions originated from these sites.

Let $A_j$ be a site of the replicated object system and
$D_L = \bigcup_{i=1}^n D_i$ be the set of data of all logical objects.
Let $O_j \subseteq D_L$. We define that site $A_j$ has an access prior-
ity to data items in $O_j$ if all transactions originated from
$A_j$ have a higher or at least equal priority to update data
items in $O_j$ than transactions originated from other sites of
the replicated object system. $O_j$ is called the access priority
set of $A_j$.

The access priority expresses the possibility of a trans-
action being committed on a site. Let $T_1 = \{c_i(p_k, d_k)\}$
and $T_2 = \{c_j(p_i, d_i)\}$ be two transactions originated from
sites $A_j$ and $d_k \in O_j$, $d_i \notin O_j$. Then $T_1$ has a greater or
equal chance to commit in $A_j$ than $T_2$.

Let $O_1$ and $O_2$ be the access priorities of site $A_1$ and
site $A_2$, respectively. We call $\sigma_{ij} = O_i \cap O_j$ the degree of
shared priority for $A_i$ and $A_j$.

The degree of shared priority shows the possibility that
a transaction partially committed on one site may conflict
with a transaction partially committed on another site. One
extreme of the degree of shared priority is that $O_j = \emptyset$ for
all $j$, which means that there is no access priority defined
for any site. Or, $\sigma_{ij} = \emptyset$ for all sites $A_i$ and $A_j$, $i \neq j$.
In that case, all access priorities are partitioned. Another
extreme is that $\sigma_{ij} = D_L$, where every site has the same
access priority, which is the total set of logical objects.

2.3 Failures

A transaction $T$ aborts if any of its calls returns a FL (e.g.,
the requested update is not allowed), or $T$ was in a PC state
and then a propagated transaction accesses the same data
items as $T$ and $T$ is then down-graded into an FL. The usual
treatment for a transaction failure is to retry it after a random
period of delay. The random delay is necessary to avoid
oscillation when two or more transactions need to use the
same data of a replica at the same time.

A site is down means that all physical objects running
on that site are down. In that case, the site cannot accept any
transactions from clients nor any propagated messages from
other sites. During the time a site is down, other sites may
have carried out some transactions. These missed transac-
tions must be executed in the same order by the failed site
when it recovers.

A network partition failure means that replicas of a ser-
vice may belong to two disconnected partitions. In this case,
the two sets of replicas should still provide the service to
clients. However, when the two partitions are re-united, the
two sets of replicas may have performed some conflicting
operations. To solve this problem, we only let the partition
with a majority of sites commit transactions. Other parti-
tions can only partially commit transactions. When the par-
titions are re-united, those committed transactions during
the partition period will be force to commit in other par-
titions. While the partially committed transactions, if they
have not been rolled back by the committed transaction from
the majority partition, will be propagated to other partitions.

For simplicity, we only assume crash failures in this pa-
per.

3 The TBIU Protocol

3.1 System Components and Their Functions

Our TBIU protocol for transaction management in a repli-
cated object system is token-based. A token circulates around
all sites in an order say, $A_1, A_2, \ldots, A_m, A_1, \ldots$. The token
also carries a timestamp for ordering transactions. Only the
token holder can propagate transactions to all sites. Other
sites without the token can only locally process the transac-
tion if the site has access priority on the transaction, result-
ing a partial commit state which may be rolled back if the
transaction is conflict with a propagated transaction from
the token holder. Each site has two queues that can only be
accessed by system components:

- A transaction request queue (TRQ), ordered by their
  local arrival time, for all incoming transactions origi-
nated from that site.
- A PC transaction queue (PCQ), ordered by the local
timestamp, for all partially committed transactions on
  that site.
The following system components running on site $A_j, j = 1, 2, \ldots, m$ are used to manage transactions in our replicated object system:

- A transaction request manager (TrM) accepts a transaction
  \[ T = \{c_1(p_1, d_1), c_2(p_2, d_2), \ldots, c_k(p_k, d_k)\} \]
  from clients. If $d_i \in O_i, for i = 1, 2, \ldots, k$, then $T$ is executed locally and put into the PCQ. Otherwise, the transaction is queued up in the TRQ.

- A token-site manager (ToM) checks if $A_i$ has the token or not. If it has, then it assigns global timestamps from the token to all PC transactions in the PC transaction queue (using its original order) and then assigns global timestamps to all transactions of the transaction request queue (using the same order). All transactions are ordered according to the global timestamps. These transactions are propagated to all sites, together with the message that the token has been assigned to the next site $A_{i+1}$ (if $m = m$ then to $A_1$). The ToM algorithm returns an OK to all partially committed transactions and returns the execution results to all transactions without a PC state.

- A non-token-site manager (NToM) has the functions of monitoring the liveliness of the token holder, requesting a new token holder, acting as the new token holder, and most importantly, accepting transactions from the token holder and processing these requests according to the order given by the token holder. If the NToM finds out that a transaction propagated from the token holder is conflict with a PC transaction processed locally, then the local PC transaction has to be aborted and removed from the PC transaction queue. Aborting a PC transaction may also affect other PC transactions.

When the logical ring has been established, each site runs the above system components as an endless loop. A client simply submits a transaction request to the TrM on a site and waits for the transaction result and the state. The returned state of a transaction at this stage can be one of OK, FL, or PC. As soon as the client receives a returned state of OK or FL, it can continue with its next operation. That means other operations, such as operations required for consistency, or for continuous committing will be dealt with by the system without the participation of the client. However, if a transaction returns PC, the client will be notified (via an asynchronous receive procedure, similar to an exception handler) later of the final state (commit or abort).

We use a Partially Committed Transaction (PCT) table, which essentially is a checking point log, to record the events of partially committed transactions in a local site. The PCT table of each site is kept in stable storage [8]. Therefore, information stored in the PCT table will not be affected by system failures. The PCT table contains information about partially committed transactions, the recovery information and the address of an asynchronous procedure used to notify the client of a transaction about the final result.

Some sites may fail when a token holder propagates its transactions to all sites. We use a failed transaction (FT) table to record all the sites and the propagated transactions that these sites have failed to execute.

### 3.2 Transaction Life-Cycle

The life-cycle of a transaction in our TBU protocol can be expressed in Figure 1. The dashed lines in Figure 1 represent asynchronous communications where the handler procedures defined in the PCT table are used.

![Figure 1: Life-cycle of a transaction](image)

A transaction $T$ is submitted by a client to the TrM of a site $A_j$. If $T$ can be executed locally, TrM will perform $T$ locally. If $T$'s local execution is successful, TrM will put $T$ into the PCQ, set up an entry for $T$ in the PCT table, and return a PC to the client immediately. If $T$'s local execution fails, an FL is returned. If $T$ cannot be executed locally, $T$ is put into the TRQ.

When $A_j$ gains the token, it uses the ToM algorithm to combine transactions in the PCQ and TRQ into a message $msg$ and sends $msg$ to all other sites. The message is ordered according to a global timestamp, with transactions in the PCQ ordered before transactions in the TRQ. The token holder also processes transactions with a PC state (i.e., in the PCQ originally) by returning an OK state to clients through the handler of each transaction stored in the PCT table. For transactions without a PC state (i.e., in the TRQ originally), the token holder will process the transactions locally using the local 2PC protocol and return the results back to clients.

When a non-token holder $A_j$ receives the above message $msg$ from the token holder, it uses the NToM algorithm to process the transactions contained in the $msg$ according to the order set by the token holder. Transactions with a PC state are executed before transactions without the PC state. All PC state transactions have to be forced to commit on $A_j$. That is, if $T \in msg, T \not= \{PC, T' \in PCQ of A_j and T'.D \cap T''.D \not= \emptyset\}$, then $T'$ has to be rolled back. Also, any PC state transactions of $A_j$ which are affected by $T'$ have to be rolled back. That is, for any $T'' \in PCQ, T''.D \cap T''.D \not= \emptyset$,
3.3 Recovery from Failures

When a site fails, the token holder detects the site failure through timeout. The failed site is then eliminated from the logical ring and the result is multicast to all members of the new ring when the token is passed to the next token holder.

When a failed site re-joins the logical ring, it multicasts the re-joining message to all members of the logical ring. The token holder then responds to this message by sending to the re-joining site its corresponding messages stored in the FAT table. The re-joining site has to perform all the transactions it missed during its down time. A new logical ring which includes the re-joined site is formed and the information is multicast to all site by the token holder when it passes the token to the next site.

When the token holder $A_i$ fails, other sites will detect that fact by a time-out since they all knew who will be the next token holder and were expecting a multicast message from it. Following [7], a message is sent to site $A_{i-1}$, asking it to be the new token holder. $A_{i-1}$ has the highest probability of being alive since it just passed the token out. $A_{i-1}$ also has the up-to-date information about the token (such as the last timestamp and the FAT entries). In the case of $A_i$'s failure, $A_{i-1}$ is the optimal choice to be the next token holder without an election. If $A_{i-1}$ fails to assume the duty as the token holder, site $A_{i-2}$ and process continues if $A_{i-2}$ fails will be selected by all alive site to be the next token holder.

When a network partition occurs, the replicated sites may be partitioned into smaller subgroups. We consider the following two situations:

- When one of the subgroups has the majority of sites. In this case, the subgroup with the majority of sites will be allowed to form a new logical ring and resume the function of propagating transactions globally. If the subgroup contains the token holder, then the token holder will be responsible for the reforming of the ring. Otherwise, a new token holder has to be decided. Transactions propagated in this subgroup will be recorded in the FAT table and then propagated to other sites when the network is recovered. Other subgroups without the majority will not be allowed to propagate their transactions. However, they are allowed to partially commit their transactions locally.

- When there is no majority in any subgroups. In this case, none of the subgroups is allowed to propagate their transactions globally. However, they are allowed to make partial commit locally for their transactions.

When the network recovers, these partially committed transactions will be processed.

4 Performance Issues

4.1 Analysis of the TBIU Protocol

In analysis of this paper, only point-to-point network is used in which a single source site sends the same message to $m$ sites by transmitting $m$ messages, one to each receiver. We assume that the communication time (including the token passing time and the transaction propagation time) of each message transmission is a constant $C$. The following parameters are used:

- $\lambda$: the arrival rate of transactions.
- $T$: the mean processing time of a transaction.
- $u$: utilisation of the transaction processing system: $u = \lambda T$.
- $p$: the probability that a transaction is queued up in the TRQ. Therefore, $1 - p$ is the probability that a transaction is to be partially committed and then queued up in the PCQ.
- $q$: the probability that a transaction is to be rolled back from the PCQ because of conflict.

We use induction to derive the time needed to process transactions in a site. Without loss of generality, let Site 1 be the token holder initially. The sequence of token passing is $1, 2, \ldots, m, 1, \ldots,$ where $m$ is the number of sites. In that case, all incoming transactions to Site 1 within a unit time will be processed and then propagated. Let $D_i$ be the transaction processing time at Site $i$, $i = 1, 2, \ldots, m$. We have:

$$D_1 = \lambda T = u.$$

For Site 2, it has to wait for $D_1$ time before it gets the token. During that time, $D_1 \lambda p$ transactions will come, of which $D_1 \lambda p = \lambda^2 T p$ of them will go into the TRQ and the cost of processing them is $\lambda^2 T p$. Similarly, $(1 - p) \lambda^2 T$ transactions will go into the PCQ. The cost of performing these transactions, then rolling back some of them, and then re-submitting these rolled back transactions can be expressed as $kq(1 - p)T^2$, where $k \geq 2$. So,

$$D_2 = \lambda T + ((\lambda T p + kq(1 - p)) \lambda T^2)$$

$$= \lambda T + (p + kq(1 - p)) \lambda T^2$$

$$= D_1(1 + \lambda T (p + kq(1 - p))).$$

Let $D_{i-1} = D_{i-2}(1 + \lambda T (p + kq(1 - p)))$, for Site $i$, we have:

$$D_i = D_{i-1} + (D_{i-1} \lambda p + kq(1 - p) D_{i-1} \lambda T)$$

$$= D_{i-1} + \lambda T D_{i-1} (p + kq(1 - p))$$

$$= D_{i-1}(1 + u(p + kq(1 - p))).$$

-890.-
By induction, we have:
\[ D_m = D_{m-1}(1 + u(p + kq(1 - p))). \]

Let
\[ D_m/D_{m-1} = (1 + u(p + kq(1 - p))) = \Delta. \]

Then
\[ D_m = \Delta D_{m-1} = \Delta^2 D_{m-2} = \cdots = \Delta^{m-1} D_1 = \Delta^{m-1} u \]
\[ = (1 + u(p + kq(1 - p)))^{m-1} u. \]

(1)

It is easy to show that if we include communication time \( C \) into the above analysis, then we have:
\[ D_m = (1 + u(p + kq(1 - p)))^{m-1} (u + C). \]

(2)

### 4.2 Comparison: Copy Token vs. TBIU

A transaction \( T \) submitted to a site \( A \) of the token copy protocol \( [9] \) enters a queue to wait for the site to become the token holder. When \( A \) becomes the token holder, \( T \) is executed locally and a result returned to the client immediately. \( T \) is packaged into a multicast message and sent to other sites for forced execution. This is a special case of the TBIU protocol. In fact, Equation (1) of Section 4.1 has two special cases.

![Figure 2: Copy token vs. TBIU](image)

* When \( p = 1 \), no partial commit is permitted. All transactions have to be put into the TRQ, waiting for the token. This is the original copy token protocol. In that case,
\[ D_m = (1 + u)^{m-1} u. \]

(3)

The TBIU protocol is better if
\[ (1 + u) > (1 + u(p + kq(1 - p))). \]

That is, if \( q < 1/k \). Figure 2 confirms this situation, where \( k = 2.5 \). The TBIU protocol is expressed as solid lines, and the copy token protocol is expressed as dots. Figure 2 shows that TBIU is better when \( q \) is small.

* When \( p = 0 \), all transactions are to be partially committed and are subject to the possibility of being rolled back. The transactions commit when the site becomes the token holder. This is a revised copy token protocol. In that case:
\[ D_m = (1 + kqu)^{m-1} u. \]

(4)

The TBIU protocol is better if
\[ (1 + kqu) > (1 + u(p + kq(1 - p))). \]

That is, when \( q > 1/k \). Figure 3 shows this situation where \( k = 2.5 \). The TBIU protocol is also expressed as solid lines, and the revised copy token protocol is expressed as dots. Figure 3 shows that TBIU is better when \( q \) is large.

![Figure 3: Revised copy token vs. TBIU](image)

### 4.3 Comparison: Independent Update vs. TBIU

Now we compare the TBIU protocol with the independent update protocol \( [4, 10] \). A transaction \( T \) submitted to a site \( A \) of the independent update protocol is executed and propagated to other sites immediately for execution. Each site \( A' \) will execute the propagated transaction and return an acknowledgment. If the execution of \( T \) on \( A' \) conflicts with other transactions already executed on \( A' \), the reconciliation process is launched, which involves the exchange of information between \( A' \) and other sites whose transactions conflict with transactions executed on \( A' \).

To simplify the analysis of the independent update protocol, we analyze the behaviors of the protocol in a unit
of time and omit other transacting the processing. We use the same eqn 4.1. Let
\(0, 1, \cdots, (m - 1)\) be the set all sites of
the system. The number of \(\mu_i\) at Site 0
(or any other site) in a time unit used to
process these transactions were \(\tau_i\) be the
conflict probability between Site 0 and Site \(i, i = 1, 2, \cdots, (m - 1)\). The re-
conciliation will be \(\mu_i \sum_{i=1}^{m-1} \tau_i\) commu-
nication is \((m - 1)\)C. So the time unit for an update protocol to process transaction unit

\[ D_i = \mu_i(1 + k \sum_{i=1}^{m-1} \tau_i) \]  

Note that normally we have rollback probability of all transactions on Site's transactions where
probability of those partially committed transactions we have
mentioned before, we assume that are not likely to conflict with other they only
access data within the access (originating site) are to be partially commit.

Intuitively, one of the major the performance of any token-based pfs the logical
ring (m). Figure 4 shows the two protocols in terms of changing on we let
\(u = 0.2, p = 0.5, k = 2.5, \) and
changes in cases of 0.1, 0.3, and 0.5 in the

![Figure 4: Independent TBU](image)

Figure 4 shows that the TBU when
\(2 \leq m \leq 13\) if \(q = \tau_i = 0.5\) 13, the
response time of the TBU protocols. However, if \(q = 0.3 < \tau_i\), the rate TBU
protocol is better becomes \(2 \leq m\) when
\(q = 0.1 < \tau_i\), then the above \(2 \leq m \\
\leq 29\). As we mentioned consider
the arrivals of transactions during process when we analysed the independ. That
is the main reason that the result update protocol is almost linear.

Even with the above analysis that the
TBU protocol is better if the number or
medium. Fortunately in practice, most replicated systems
will have a small number of replicas.

5 Summary

What we have achieved in this paper is a protocol that allows
independent update locally. These locally performed updates are propagated when a site becomes the token holder.
All propagated update transactions have a higher priority
than any locally performed transactions and will be committed.
Those locally performed transactions whose updates
conflict with any propagated updates will be rolled back locally. The share priority mechanism is defined to control
the number of independent updates in a site.

References


[2] D. Babbanara and H. Garcia-Molina. The case for con-
trolled inconsistency in replicated data. In Proceedings
of the 1st Workshop on Management of Replicated

exploring the semantics of distributed services. In
Proceedings of the 1st Workshop on Management of Replicated

rat. Independent updates and incremental agreement in
replicated databases. Journal of Parallel and Dis-
tributed Databases, 3(3), July 1995.

[5] P. A. Bernstein and E. Newcomer. Principles of Trans-
action Processing for the System Professionals. Mor-

lease procedure call transactions. The Journal of Sys-

sis of an efficient and reliable atomic multicast protoc.

Notes in Computer Science, volume 105, pages 246–
265. Springer-Verlag, Berlin, Germany, 1981.

[9] T. Minoura and G. Wiederhold. Resilient ext-
tended two-copy token scheme for a distributed database. IEEE Transactions on Software Engineer-

[10] S. Ceri, M. A. W. Houstma, A. M. Keller, and P. Sam-
rat. The case for independent updates. In IEEE Work-
shop on Management of Replicated Data II, November
A MODEL OF EXECUTION TIME ESTIMATING FOR RPC-ORIENTED PROGRAMS

Wanlei Zhou
Brian Molinari
Department of Computer Science
The Australian National University
Canberra, ACT 2601, Australia
e-mail: zhou@anu.edu.au

ABSTRACT

This paper describes an execution time estimating model for programs which use Remote Procedure Calls (RPCs) as the tool for distributed computing. At first a general model with no closed-form solution is developed and a nondeterministic algorithm for its solution is given. Then the closed-form solution for a special case of the general model is derived. After using some approximate methods, the lower and upper bounds of the general form solution are described. The last section of this paper presents two examples of the application of the model.

Key Words: Concurrency, Distributed computing, Remote procedure call, Performance evaluation.

1. INTRODUCTION

The Remote Procedure Call (RPC) is a powerful primitive for distributed programming, and the growing interest in its use demands tools for modeling and analyzing the performance of such programs. Because a remote procedure call blocks the calling process until the call is completed and a reply has been received, a concurrency primitive such as COBEGIN or FORK is usually used to introduce the parallelism into the program. In this article, we consider the COBEGIN-C OEND primitive with the usual semantics.

We term an RPC-oriented program (in short, an RPC program) as one which executes on a local host and which calls, both separately and concurrently, remote procedures located on other hosts.

One of the successful commercial RPC-oriented distributed computing tools is the Network Computing System (NCS) (NCS and Network Computing System are trademarks of Apollo Computer Inc.). In this system, an RPC program consists of two kinds of programs: the server program and the client program. Usually server programs reside on remote hosts, while the client program resides on the local host (relative to the user). The user uses the client program to access the functions provided by the server programs. Also, a program must register all of its services (remote procedures) with Location Brokers. The client program can then find the service by interrogating the Location Brokers. After the client finds the location of the service, it then calls the service directly.

Usually a client program knows nothing about the location of the RPCs, and the only way to get them is through the Location Brokers during program execution. Also, the location of a remote procedure may be changed by its server or by some application programs during the execution. So, without loss of generality, we will assume that no locations of remote procedures are known by an RPC program before its execution. In that case, all remote procedure calls within an RPC program are channeled through Location Brokers. The operational semantics of an RPC program is indicated in the following figure.
The program interrogates the location broker for the identity of the server for a remote procedure call. The call is then placed on a queue associated with the server.

One of the important performance metrics for a concurrent program is the execution time. Many existing articles discussing the execution time estimation of concurrent programs are based on queuing theory. Heidelberg and Trivedi discussed analytic queuing models for programs with internal concurrency. Thomasian and Bay presented several queuing network models which may be used to analyze parallel processing of task systems. The queuing network model given by Almeida and Dowdy can be used to analyze the performance of programs with concurrency/synchronization schemes.

Although queuing theory is a powerful tool in the analysis of concurrent models, queuing network models with closed form solution are not directly applicable to these systems because of the internal program concurrency. Also, because of the huge number of states, models with non-closed form solution are often not feasible. Another thing is, the queuing models obtain the execution time from the system's viewpoint by estimating all the possible jobs. While a user is often interested in the execution time of his/her own job, that is, from the user's viewpoint. Based on the user's viewpoint, we present in this article an execution time evaluation model which has closed form solution for simple RPC programs and non-deterministic algorithm as well as upper and lower bounds for complex RPC programs.

2. A MODEL OF RPC PROGRAMS

2.1. Syntax Issues

An RPC program executing on the local host may do several things: At first, it may execute program segments which are completely located on the local host; secondly, it may call several remote procedures in sequence; thirdly, it may call several remote procedures in parallel. Generally speaking, the execution time of a remote procedure call is much longer than the execution time of a local procedure call because the RPC will involve some remote communications. To analyze an RPC program, we will omit the program segments executed on the local host except for the segments which may be related to the concurrent control of RPCs. That is, we idealize RPC programs to contain only RPCs and some necessary control stuff, and consider all local execution of the program as zero time. The motivation for that is the fact we want to study programs that are dominated by the time spent in RPCs.

A sequential RPC program block is indicated by

\[
\text{BEGIN} \quad a \quad b \quad c \quad \text{END} \quad (2.1)
\]

where \(a\), \(b\), and \(c\) are atomic remote procedures (or simply, atoms). That is, no remote procedures are called again from these procedures. Sequential RPC program blocks offer no speedup in a distributed system, because of the remote procedure call semantics. The execution time of a sequential program block is the sum of the execution times of \(a\), \(b\) and \(c\). Our model allows remote procedure calls to be made
concurrently, as indicated by

\[
\text{COBEGIN} \quad \text{d; e; f} \quad \text{COEND}
\] (2.2)

If the atomic procedures \(d, e, \) and \(f\) are supported on different server hosts then they can be executed in parallel and the execution time of the \text{COBEGIN} \ldots \text{COEND} block is simply that of the largest component. But usually these remote procedures are allocated by the location broker to a set of available hosts. This means the evaluation of execution time will not be as such simple.

Our model is concerned with programs constructed from a set of atomic remote procedures by the repeated application of the above two operators. The abstract syntax of these programs is quite simple.

\[
\text{prog ::= seq\_block | par\_block | atom}
\]

\[
\text{seq\_block ::= BEGIN \{ prog \} END}
\] (2.3)

\[
\text{par\_block ::= \text{COBEGIN} \{ prog \} \text{COEND}}
\]

Next is an example of an RPC program (where \(A_i\) and \(A_j\) are atoms). Its motivation is given in section 4.2.

\[
\text{BEGIN}
\]

\[
A_1 \quad \text{COBEGIN}
\]

\[
\quad \text{BEGIN } A_2 ; \text{COBEGIN } A_{21} ; A_{22} \text{COEND END;}
\]

\[
A_3 ;
\]

\[
\quad \text{BEGIN } A_4 ; \text{COBEGIN } A_{41} ; A_{42} ; A_{43} ; A_{44} \text{COEND END}
\]

\[
A_5 \quad \text{COEND}
\]

\[
\text{END}
\]

Equivalently we can associate each program with a flowgraph. It is built up from its atomic remote procedure calls, with edges denote the atomic procedures and the nodes denote the begin and end of sequential atoms and \text{COBEGIN}-\text{COEND} operations. Two special nodes \(s\) and \(f\) denote the begin and end of the program, respectively. The flowgraph of the previous RPC program is given in figure 3. It is easy to see that the flowgraph of an RPC program is acyclic.

### 2.2. Semantics Issues

First we shall formalize the situation indicated in figure 1. If we denote \(P\) as the set of remote procedure atoms and \(N\) as the set of server hosts, then we can represent the location broker by a function \(m : P \rightarrow N\). Further, let us model the execution time of a remote procedure by a function \(\text{time} : P \rightarrow \text{real}\). Given a remote procedure \(p \in P\), the location broker tells us the host \(m(p)\), and the execution of \(p\) will take time \(\text{time}(p)\).

The operational semantics of figure 1 shall be made precise by describing an algorithm which determines the execution time of an RPC program. This algorithm consists of a "colouring game" on the associated flowgraph \(g\). We use the following colour scheme for flowgraph arcs:

- **white**: initial condition
- **blue**: in RPC queue
- **red**: executing
- **black**: completed

and all arcs progress through the colours in this order. The game has the invariant: *All red arcs are associated with a different server*. Further, we say that a blue arc can execute if no red arc is associated with the same server.

The game involves assigning a value to an attribute \text{CompletionTime} for each node and each arc of the flowgraph. It also involves a single global variable, denoted \text{CurrentTime} (with initial value 0).

**WHILE** red arcs remain **DO**
select red arc \( x \) with minimum completion time;
colour \( x \) black;
\( \text{CurrentTime} := x \text{CompletionTime} \);
denote target node of \( x \) by \( n \);
IF all arcs \( y \) entering \( n \) are black THEN
\( n \text{CompletionTime} := \max(y \text{CompletionTime}) \); (2.4)
colour blue all the arcs leaving \( n \);
ENDIF
WHILE there exist blue arcs which can execute DO
select a blue arc \( v \) which can execute;
colour \( v \) red;
\( v \text{CompletionTime} := \text{CurrentTime} + v \text{ExecutionTime} \);
ENDWHILE
ENDWHILE

The initialization includes assigning \( \text{CompletionTime} = 0 \) at the start node and colouring blue all arcs leaving it. These blue arcs are then processed according to the last statement of the algorithm loop.

There are two sources of nondeterminism in the algorithm, corresponding to the selection operations. The second of these is the most significant, corresponding to the scheduling of the queues in figure 1. That the execution time depends on the selection strategy is easy to see. A simple example is provided by the following flowgraph

![Flowgraph](image)

where \( a, b, \) and \( c \) are atomic remote procedures with unit execution time. Consider \( a \) and \( b \) to be mapped to host 1 and \( c \) to be mapped to host 2. If procedure \( a \) is selected first the overall execution time is 3 units, while if procedure \( b \) is selected first the overall execution time is 2 units (\( a \) and \( c \) can execute in parallel).

2.3. Performance Measures

Given a location broker map \( m \), an RPC program \( g \) will execute (according to our model) in a time denoted \( T_{\alpha}(g) \). This time depends on the topology of the flowgraph of \( g \), and the scheduling of the atoms over hosts. Our problem interest is that repeated executions may take place, with a different location broker map each time. If we denote by \( M \) the set of maps of interest, then the standard metrics are

\[
I_{\min}(g) = \min_{m \in M} T_{\alpha}(g); \quad I_{\max}(g) = \max_{m \in M} T_{\alpha}(g);
\]

\[
I_{\alpha}(g) = \exp \left( T_{\alpha}(g) \right) = \sum_{m \in M} T_{\alpha}(g) p_r(m).
\] (2.5)

where \( p_r \) is a probability function defined over the set of maps \( M \). We are only interested in the equiprobable case, namely,

\[
p_r(m) = \frac{1}{S}; \quad \text{where } S = \text{card}(M).
\]

3. PARALLEL BLOCK SPECIAL CASE

3.1. Analysis

In this section we consider the special case of programs involving the parallel execution of atomic procedure calls. These programs have the structure indicated in equation (2.2), and equivalently have a flowgraph of the following form (where \( p_i \) are atoms).
Now given \( n : N \), we denote by \( m^{-1}(n) \) the set of remote procedures that execute on the host \( n \). If we denote by \( P \) the set of all remote procedures involved in the graph (3.1), then \( A_n = P \cap m^{-1}(n) \) is the subset of \( P \) that will execute on \( n \). In Fig. 1 these are the remote procedures that will be placed on the queue of the server \( n \). To get a closed-form solution we need to make a further assumption, namely, that all procedures execute in a standard time.

\[
T(p) = 1 \quad \text{for all } p \in P
\]

(3.2)

The analysis is now simple. The time needed to execute the queue on node \( n \) is the sum of the queue components

\[
T_n(A_n) = \sum_{p \in A_n} T(p)
\]

(3.3)

while the time needed to execute all queues is the time of the largest queue.

\[
T_n(e) = \max_{a \in P} T_n(A_a)
\]

(3.4)

Because of (3.2), (3.3) can be written as

\[
T_n(A_n) = \text{card} (A_n)
\]

(3.5)

3.2. Closed Form Solution

We characterise the set \( M \) of maps as follows. Each \( m : M \) is a map from the finite set \( P \) to the finite set \( N \) where we denote

\[
k = \text{card} (N) ; \quad L = \text{card} (P).
\]

(3.6)

When maps are being set up in the location broker the procedures themselves are not distinguished. It is their allocation to a host that is of interest. Hence we identify a distinct map with the allocation of \( k \) identical "balls" to \( L \) different "boxes". This is a standard problem in combinatorics and its solution is well-known\(^7\). The maximum and minimum measures are easy:

\[
T_{\max} (e) = k ; \quad T_{\min} (e) = \left[ \frac{k}{L} \right]
\]

(3.7)

The first formula assumes to the case where all procedures are mapped to a single host, while the second corresponds to the case where the procedures are distributed as evenly as possible. Here \( \lfloor x \rfloor \) denotes the integer least upper bound.

The average measure is harder. First we compute \( S \), the total number of maps. From combinatorics results\(^7\) we know this is a type-2 distribution problem and

\[
S = \binom{L}{k} \binom{L, \infty} = \left[ \frac{L+k-1}{L-1} \right]
\]

(3.8)

where \( [x, \infty) \) is the restrict condition of balls in a single box. But from the problem we know the maximum balls within a single box can not be more than \( k \). So here the restrict condition can be changed to \([0, k)\).

If we denote by \( \mathcal{Q}_i^{\mu L} \) the number of maps with \( \text{max card} (A_n) = i \), then

\[
T_n(e) = \frac{1}{S} (1 \times \mathcal{Q}_1^{\mu L} + 2 \times \mathcal{Q}_2^{\mu L} + \cdots + k \times \mathcal{Q}_k^{\mu L})
\]

(3.9)
The combinatorial meaning of \( Q^a_{iL} \) is: \( k \) identical balls are to be distributed to \( L \) different boxes and the maximum number of balls within a single box is exactly equal to \( i \) and at least one box has this number of balls. If the maximum number of balls within a single box is less than or equal to \( i \), then

\[
d^a_k (L, [0, i]) = \sum_{0 \leq j \leq L} (-1)^j j! \frac{(L-k-(j+1)i-j)!}{L-i}
\]  

(3.10)

So,

\[
Q^a_{iL} = d^a_k (L, [0, i]) - d^a_k (L, [0, i-1]),
\]

\( i = 1, 2, \ldots, k, \) and \( d^a_k (L, [0, 0]) = 0 \).

It is evident that \( S = \sum_{i=1}^{k} Q^a_{iL} \).

4. LOWER AND UPPER BOUNDS ESTIMATION

4.1. Lower Bound

At first we define an extended parallel block as

\begin{align*}
B_1 & : B_2 : \cdots : B_k \\
& \text{(4.1)}
\end{align*}

where \( B_i \) is a parallel block (as in (2.2)) or an extended parallel block. This is a recursive definition. If there are \( L \) hosts, the average execution time of an extended parallel block \( A \) is defined as:

\[
\frac{1}{L} \sum_{i=1}^{k} T_{lev}(B_i) \\
= \max_{i=1}^{k} \left( T_{lev}(B_i) \right)
\]

(4.2)

If \( B_j \) is a parallel block, then \( T_{lev}(B_j) \) is calculated by using the formula in section 3. That is, in that case we have \( T_{lev}(B_j) = T_{lev}(B_j) \). If \( g \) is an RPC program, then the calculation can be carried out from the inner parallel blocks of \( g \) to the outer, and we have,

\[
T_{lev}(g) \leq T_{lev}(A) \leq T_{lev}(g)
\]

(4.3)

where \( T_{lev}(g) \) is defined in (3.7).

The left part of (4.3) is evident because \( T_{lev}(g) \) considers some sequential allocation of atoms in parallel blocks of program \( g \) while \( T_{lev}(g) \) only considers the even allocation of atoms. The right part of (4.3) is also true because: (1) In (4.1), if \( k = 1 \), then \( T_{lev}(A) = T_{lev}(A) \). (2) If \( k > 1 \) and \( B_j \) and \( B_j \) are two parallel blocks, then the calculation of (4.2) considers them independently, while in fact they may execute concurrently. In that case we have \( T_{lev}(A) \leq T_{lev}(A) \). (3) The meaning of (4.3) is to consider \( k \) sub-extended parallel blocks as being allocated on \( L \) hosts as evenly as possible, while it is not always the case. So we still have \( T_{lev}(A) \leq T_{lev}(A) \). This proves that (4.3) holds.

4.2. Upper Bound

If \( g \) is an RPC program and \( G \) is its flowgraph, we define the level of an atom \( p : g \) as the length of the path from node \( p \) to the end node of \( g \) and denote it as \( p \), level. Now the upper bound of \( g \)'s average execution time \( T_{lev}(g) \) can be calculated as follows:

(1) Construct parallel block \( A_1 \) such that if \( p : A_1 \), then \( p \), level = \( i \), \( i = 1, \ldots, M \) and \( M \) is the maximum level of \( G \);

(2) Calculate \( T_{lev}(A_i) \) by using formula in section 3;
(3) \[ T_{ua}(x) = \sum_{i=1}^{n} T_{ui}(A_i). \]

It is not difficult to see that
\[ T_{ar}(x) \leq T_{ua}(x) \leq T_{max}(x) \tag{4(A)} \]

The right part of (4A) is evident because \( T_{max}(x) \) views all atoms as sequentially executed, while \( T_{ua}(x) \) considers some degree of parallelism. The left part is also true because atoms in different levels may execute concurrently, while \( T_{ar}(x) \) views them as strictly sequential execution. So (4A) holds.

5. APPLICATIONS OF THE MODEL

5.1. A Simple Example: The Seller-Buyer System

Suppose there is one Seller and several Buyers. The Seller at first sends a message to each Buyer by using a remote procedure provided by the Buyer, describing the price, amount and performance of some goods. After a Buyer receives the message, he will decide whether or not to buy the goods and what are the amount and bid price he is going to offer. When the Seller receives the bids from the return values of the remote procedure, he will send out goods to the Buyer with the best offer, again using a remote procedure provided by the Buyer. The Seller resides on the main host, while the buyers are allocated on the distributed system (which consists of 5 hosts). The structure of the RPC program \( g \) corresponds to the flowgraph of figure 2, where \( A_1 \) to \( A_7 \) are remote procedures provided by Buyer 1 to 5 which can be issued by the Seller to send out the message and get the return bids, and \( A_g \) is the remote procedure call the Seller sends out goods to the best Buyer. Remote procedure call \( A_1 \) is used by the Seller to do some preparation before issuing the message.

![Flowgraph of the Simple Example](image)

Suppose all of the \( A_g \)'s (i=1 to 7) are atomic remote procedure calls and the execution times for \( A_1 \) to \( A_7 \) are all 1. For convenience, we denote the parallel block of the program as \( C \). We have
\[
T_{ar}(C) = \frac{1}{S} \sum_{i=1}^{S} t \times Q_i^{5.5}
\]
\[
= \frac{1}{126} \times (1 \times 1 + 2 \times 5 + 3 \times 5 + 4 \times 20 + 5 \times 5) = \frac{356}{126} = 2.8.
\]

That is, the execution time for \( C \) is 2.8 time unit. So the evaluation time of this RPC program is \( T_{ar}(C) = 1 + 2.8 + 1 = 4.8 \) time units.

Now, if we assume the distributed system has only 3 hosts instead of 5, then
\[
T_{ar}(C) = \frac{1}{3} \sum_{i=1}^{S} t \times Q_i^{5.5}
\]
\[
= \frac{1}{21} \times (1 \times 1 + 2 \times 3 + 3 \times 9 + 4 \times 6 + 5 \times 3) = \frac{72}{21} = 3.4.
\]
So the estimated execution time of this RPC program will be $5.4$ time units. In each case the lower bound and upper bound of the average execution time is the same as the average execution time.

5.2. A More Complex Example: Extended Seller-Buyer System

![Diagram](image)

Fig. 3. Structure of the Complex Example

Next we extend the above Seller-Buyer system by adding two wholesalers. That is, the Seller concurrently sends out the goods information to some Buyers and two wholesale-persons. These two persons then send the information to their own customers (Buyers) with their own comments which may influence these Buyers. All the bids return to the Seller. At last, the Seller will send the goods to the Buyer with the best offer. The structure of RPC program $g$ corresponds to the flowgraph of Figure 3(a). Here $A_1$ and $A_2$ are the same as the $A_1$ and $A_7$ of the last example, respectively. $A_2$ is the remote procedure of the first wholeseller, while $A_{21}$ and $A_{22}$ are its customers' remote procedure calls. $A_{31}$ is the RPC of another wholeseller and $A_{41}$ are the RPCs of the customers. $A_5$ is the Seller's customer. Here $A_j$ and $A_{ij}$ are atoms and their completion time are all $1$ unit.
To this time the derivation of a closed form solution to $T_{m}(g)$ for this program is intractable. Instead we shall indicate the use of lower and upper bounds to estimate the average execution time of $g$.

Fig. 3(b) is the restructuring of Fig. 3(a) using extended parallel blocks, and Fig. 3(e) is the restructuring of Fig. 3(c) using levels. If we assume that there are 5 hosts, then we have

$$T_{La}(g) = 1 + \max \left( \frac{1+3+1+(1+2.4)}{5}, 3.4 \right) + 1 = 5.4,$$

$$T_{La}(g) = 1 + 1.9 + 3.3 + 1 = 7.2.$$

So, $5.4 \leq T_{av}(g) \leq 7.2$. If there are 3 hosts, then we have $5.8 \leq T_{av}(g) \leq 8.2$.

6. CONCLUSIONS

We have presented a performance evaluation model for RPC programs. Then we mentioned a non-deterministic algorithm for general case solution. The closed-form solution of the general case is infeasible instead, we present a closed-form solution for a special case -- the parallel blocks. Based on this, the lower and upper bounds of the general form solution are given and two examples are described.

References

A PERFORMANCE EVALUATION MODEL FOR PROGRAMS USING REMOTE PROCEDURE CALLS

Wanlei Zhou and Brian Molinari
Department of Computer Science
The Australia National University
ACT 2601

ABSTRACT

This article describes a performance evaluation model for programs which use RPCs (Remote Procedure Call) as the tool for distributed computing. At first a general model with no closed-form solution is developed and a nondeterministic algorithm for its solution is given. Then the closed-form solution for a special case of the general model is derived. The last section of this article presents two examples of the application of the model.

Key Words: Concurrency; Distributed programming; Remote procedure call; Networks; Performance evaluation.

CR Categories: C.2.4; C.4; D.1.3.

1. INTRODUCTION

The Remote Procedure Call (RPC) is a powerful primitive for distributed programming (Nelson (1981), Gifford (1986)), and the growing interest in its use demands tools for modeling and analysing the performance of such programs. Because a remote procedure call blocks the calling process until the call is completed and a reply has been received, a concurrency primitive such as COBEGIN or FORK is usually used to introduce the parallelism into the program. In this article, we consider the COBEGIN-COEND primitive with the usual semantics. We term an RPC program as one which executes on a main host and which calls, both separately and concurrently, remote procedures located on other hosts.

One of the successful commercial RPC-oriented distributed computing tools is the Network Computing System (NCS)1 (Apollo (1987), Zhou (1988)). In this system, a server program must register all of its services (remote procedures) with Location Broker. The client program can then find the service by interrogating the Location Broker. After the client finds the location of the service, it then calls the service directly.

For some specific applications, the client program knows definitely where the RPCs are located. It then can call them directly instead of through the Location Broker. But in the general situation, a client program knows nothing about the location of the RPCs, and the only way to get them is through the Location Broker during program execution. Also, the location of a remote procedure may be changed by its server or by some application programs during the execution. So, without lost of generality, we will assume that no locations of remote procedures are known by an RPC program before its execution. So, all remote procedure calls within an RPC program are channeled through Location Brokers. The operational semantics of an RPC program is indicated in the following figure.

---

1 NCS and Network Computing System are trademarks of Apollo Computer Inc.
The program interrogates the location broker for the identity of the server for a remote procedure call. The call is then placed on a queue associated with the server.

Two methods can be used in the performance evaluation of such programs. One is to set up a testbed and evaluate the performance by experiment and simulation; the other is to abstract the real system to an analytic model and evaluate the performance by model solution. We chose the later method for its simplicity and economy.

Many articles discussed the performance evaluation of concurrent programs. Heidelberg and Trivedi (1983) discuss analytic queueing models for programs with internal concurrency. Thomasian and Bay (1983) presented several queueing network models which may be used to analyze parallel processing of task systems. The queueing network model given by Almeida and Dowdy (1986) can be used to analyze the performance of programs with concurrency/synchronization schemes. Although queueing theory is a powerful tool in the analysis of concurrent models, queueing network models with closed form solution are not directly applicable to these systems (Chandy and Martin, 1983) because of the internal program concurrency. Also, because of the huge number of states, models with non-closed form solution are often not feasible.

In this article we present a performance evaluation model which has closed form solution for simple RPC programs and non-deterministic algorithm for complex RPC programs.

2. A MODEL OF RPC PROGRAMS

2.1. Syntax Issues

As mentioned earlier, an RPC program executing on the main host may do several things: first, it may execute program segments which are completely located on the main host; second, it may call several remote procedures in sequence; third, it may call several remote procedures in parallel. Generally speaking, the execution time of a remote procedure call is much longer than the execution time of a local procedure call because the RPC will involve some remote communications. To analyze an RPC program, we will omit the program segments executed on the main host except for the segments which may be related to the concurrent control of RPCs. That is, we idealize RPC programs to contain only RPCs and some necessary control stuff. The motivation for that is the fact we want to study programs that are dominated by the time spent in RPCs. Also, we assume that every remote procedure call in an RPC program finds its destination server (host) through the Location Broker and each host has only one server.

A sequential RPC program, indicated by

\[
\text{BEGIN}\nonumber
\begin{align}
&\text{a;} \\
&\text{b;} \\
&\text{c} \\
&\text{END}
\end{align}
\tag{1}
\]

where a, b, and c are RPCs, offers no speedup in a distributed system, because of the remote procedure call semantics. Our model allows remote procedure calls to be made concurrently, as indicated by
COBEGIN
    d;
    e;
    f
COEND

If the procedures d, e, and f are supported on different server hosts then they can be executed in parallel and the execution time of the COBEGIN ... COEND block is simply that of the largest component. But usually these atoms are allocated by the location broker to a set of available processors. This means the evaluation of execution time will not be as such simple.

Our model is concerned with programs constructed from a set of atomic remote procedures by the repeated application of these two operators. The abstract syntax of these programs is quite simple.

\[
\text{prog ::= seq\_block \mid par\_block \mid atom}
\]

\[
\text{seq\_block ::= "BEGIN" ( prog ) "END"}
\]

\[
\text{par\_block ::= "COBEGIN" ( prog ) "COEND"}
\]

An example of RPC program is as follows (where \(A_i\) and \(A_{ij}\) are atoms). Its motivation is given in section 4.2.

BEGIN
    \(A_1\)
    COBEGIN
        BEGIN \(A_2\); COBEGIN \(A_{21}; A_{22}\) COEND END;
        \(A_3\);
        BEGIN \(A_4\); COBEGIN \(A_{41}; A_{42}; A_{43}; A_{44}\) COEND END
    COEND
    \(A_5\)
END

Equivalently we can associate each program with a flowgraph (Marcotty and Ledgard, (1986)), indicated by

\[
\text{(4)}
\]

where \(s\) and \(f\) are denote start and finish nodes, respectively. The flowgraph of a program is built up from its atomic remote procedure calls, denoted by

\[
\text{(5)}
\]

By repeated application of the sequential and parallel operators, indicated by

sequence

parallel
The flowgraph of the previous RPC program is given in figure 3.

2.2. Semantics Issues

First we shall formalise the situation indicated in figure 1. If we denote

\[ P \rightarrow \text{set of remote procedure atoms} \]
\[ N \rightarrow \text{set of server hosts} \]

then we can represent the location broker by a function \( m : P \rightarrow N \). Further, let us model the execution time of a remote procedure by a function \( t : P \rightarrow \text{reals} \). Given a remote procedure \( p : P \), the location broker tells us the host \( m(p) \), and the execution time will take time \( t(p) \). We assume for simplicity that the executions of all remote procedures are limited to their hosts. That is, within a remote procedure one cannot call any other remote procedure.

The operational semantics of figure 1 shall be made precise by describing an algorithm which determines the execution time of an RPC program. This algorithm consists of a "colouring game" on the associated flowgraph \( g \). We use the following colour scheme for flowgraph arcs:

- **white**: initial condition
- **blue**: in RPC queue
- **red**: executing
- **black**: completed

and all arcs progress through the colours in this order. The game has the following invariant:

All red arcs are associated with a different server.

Further, we say that a blue arc can **execute** if no red arc is associated with the same server.

The game involves assigning a value to an attribute \( \text{CompletionTime} \) for each node and each arc of the flowgraph. It also involves a single global variable, denoted \( \text{CurrentTime} \) (with initial value 0).

```
WHILE red arcs remain DO
    select red arc \( x \) with minimum completion time;
    colour \( x \) black;
    \( \text{CurrentTime} := x.\text{CompletionTime} \);
    denote target node of \( x \) by \( n \);
    IF all arcs \( y \) entering \( n \) are black THEN
        \( n.\text{CompletionTime} := \max(y.\text{CompletionTime}) \);
        colour blue all the arcs leaving \( n \); (6)
    ENDIF
ENDWHILE
```

The initialisation includes assigning \( \text{CompletionTime} = 0 \) at the start node and colouring blue all arcs leaving it. These blue arcs are then processed according to the last statement of the algorithm loop.

There are two sources of nondeterminism in the algorithm, corresponding to the selection operations. The second of these is the most significant, corresponding to the scheduling of the queues in figure 1. That the execution time depends on the selection strategy is easy to see. A simple example is provided by the following flowgraph
where \( a, b, \) and \( c \) are atomic remote procedures with unit execution time. Consider \( a \) and \( b \) to be mapped to server 1 and \( c \) to be mapped to server 2. If procedure \( a \) is selected first the overall execution time is 3 units, while if procedure \( b \) is selected first the overall execution time is 2 units (\( a \) and \( c \) can execute in parallel).

2.3. Performance Measures

Given a location broker map \( m \), an RPC program \( g \) will execute (according to our model) in a time denoted \( T_m(g) \). This time depends on the topology of the flow graph of \( g \), and the scheduling of the atoms over hosts. Our problem interest is that repeated executions may take place, with a different location broker map each time. If we denote by \( M \) the set of maps of interest, then the standard metrics are

\[
T_{\text{min}}(g) = \min_{m \in M} T_m(g) \quad T_{\text{max}}(g) = \max_{m \in M} T_m(g) \quad T_{\text{av}}(g) = \exp(T_m(g)) = \sum_{m \in M} T_m(g)pr(m) \quad (7)
\]

where \( pr \) is a probability function defined over the set of maps \( M \). We are only interested in the equiprobable case, namely,

\[
pr(m) = \frac{1}{S} \quad \text{where } S = \text{card}(M).
\]

3. PARALLEL BLOCK SPECIAL CASE

3.1. Analysis

In this section we consider the special case of programs involving the parallel execution of atomic procedure calls. These programs have the structure indicated in equation (2), and equivalently have a flow graph of the following form (where \( p_i \) are atoms).

![Flow graph example](image)

Now given \( n : N \), we denote by \( m^{-1}(n) \) the set of remote procedures that execute on the host \( n \). If we denote by \( P \) the set of all remote procedures involved in the graph (8), then \( A_n = A \cap m^{-1}(n) \) is the subset of \( P \) that will execute on \( n \). In Fig. 1 these are the remote procedures that will be placed on the queue of the server \( n \).

To get a closed-form solution we need to make a further assumption, namely, that all procedures execute in a standard time.

\[
t(p) = 1 \quad \text{for all } p \in P \quad (9)
\]

The analysis is now simple. The time needed to execute the queue on node \( n \) is the sum of the queue components

\[
T_m(A_n) = \sum_{p \in A_n} t(p) \quad (10)
\]

while the time needed to execute all queues is the time of the largest queue,

\[
T_m(g) = \max_{n \in N} T_m(A_n) \quad (11)
\]

Because of (9), (10) can be written as

\[
T_m(A_n) = \text{card}(A_n) \quad (12)
\]
3.2. Closed Form Solution

We characterise the set $M$ of maps as follows. Each $m: M$ is a map from the finite set $P$ to the finite set $N$ where we denote

$$ k = \text{card}(N); \quad L = \text{card}(P). $$

When maps are being set up in the location broker the procedures themselves are not distinguished. It is their allocation to a host that is of interest. Hence we identify a distinct map with the allocation of $k$ identical "balls" to $L$ different "boxes". This is a standard problem in combinatorics and its solution is well-known (Bogart (1983), Ke and Wei (1981)).

The maximum and minimum measures are easy:

$$ T_{\text{max}}(q) = k; \quad T_{\text{min}}(q) = \left\lfloor \frac{k}{L} \right\rfloor. $$

The first formula assumes to the case where all procedures are mapped to a single host, while the second corresponds to the case where the procedures are distributed as evenly as possible. Here $\left\lfloor \frac{x}{k} \right\rfloor$ denotes the integer least upper bound.

The average measure is harder. First we compute $S$, the total number of maps. From combinatorics results (Bogart (1983), Ke and Wei (1981)), we know this is a type-2 distribution problem and

$$ S = d^P(L, [0, \infty)) = \binom{L + k - 1}{L - 1} $$

where $(0, \infty)$ is the restrict condition of balls in a single box. But from the problem we know the maximum balls within a single box can not be more than $k$. So here the restrict condition can be changed to $[0, k]$.

If we denote by $Q^P_{i\ell}$ the number of maps with

$$ \max_{\alpha \in N} \text{card}(A_{\alpha}) = i $$

then

$$ T_a(a) = \frac{1}{S} (1 \times Q^P_{1L} + 2 \times Q^P_{2L} + \cdots + k \times Q^P_{kL}) = \frac{1}{S} \sum_{i=1}^{i=k} i \times Q^P_{iL}. $$

The combinatorial meaning of $Q^P_{i\ell}$ is: $k$ identical balls are to be distributed to $L$ different boxes and the maximum number of balls within a single box is exactly equal to $i$ and at least one box has this number of balls. From Ke and Wei (1981), if the maximum number of balls within a single box is less than or equal to $i$, then

$$ d^P(L, [0, i]) = \sum_{0 \leq j \leq i} (-1)^j \binom{L + k - i + 1}{L - 1} $$

So,

$$ Q^P_{i\ell} = d^P(L, [0, i]) - d^P(L, [0, i-1]), \quad i = 1, 2, \cdots, k, \text{ and } d^P(L, [0, 0]) = 0. $$

It is evident that

$$ S = \sum_{i=1}^{i=k} Q^P_{i\ell}. $$

Example: Suppose we have an RPC program $A$ with three atoms grouped as (8) and that there are 3 hosts in the system to execute the RPCs. In this case $L = 3$ and $k = 3$. Direct evaluation of equation (15) and (18) produce

$$ S = \binom{3 + 3 - 1}{3 - 1} = \binom{5}{2} = 10, \quad \text{and} $$

$$ Q^{P3}_{13} = 1, \quad Q^{P3}_{23} = 6, \quad Q^{P3}_{33} = 3. $$

This can be illustrated by representing the set $M$ of maps by the following signatures.
(3, 0, 0) (0, 3, 0) (0, 0, 3)  
(2, 1, 0) (2, 0, 1) (0, 2, 1) (1, 2, 0) (0, 1, 2) (1, 0, 2)  
(1, 1, 1)

where the i-th item of each signature gives the number of procedures mapped to the i-th host. The first row of signatures corresponds to $Q_3^{1/3}$, the second to $Q_2^{1/2}$ and the third to $Q_1^{1/5}$. Now we can get its execution time evaluation:

$$E_A = \frac{1}{5} \sum_{i=1}^{5} i \times Q_i^{1/5} = \frac{1}{10} \times (1 \times 1 + 2 \times 6 + 3 \times 3) = \frac{22}{10} = 2.2.$$  
That is, the execution time for $A$ is 2.2 time units.

4. APPLICATIONS OF THE MODEL

4.1. A Seller-Buyer System

Suppose there is one Seller and several Buyers. The Seller at first sends a message to each Buyer using a remote procedure provided by the Buyer, describing the price, amount and performance of some goods. After a Buyer receives the message, he will decide whether or not to buy the goods and what are the amount and bid price he is going to offer. When the Seller receives the bids from the return values of the remote procedure, he will send out goods to the Buyer with the best offer, again using a remote procedure provided by the Buyer. The Seller resides on the main host, while the buyers are allocated on the distributed system (which consists of 5 hosts) The RPC program structure corresponds to the flowgraph of figure 2, where $A_2$ to $A_6$ are remote procedures provided by Buyer 1 to 5 which can be used by the Seller to send out the message and get the return bids, and $A_7$ is the remote procedure call the Seller sends out goods to the best Buyer. Remote procedure call $A_1$ is used by the Seller to do some preparation before issuing the message.

![Flowgraph of the Seller-Buyer System](image)

Fig. 2. Flowgraph of the Seller-Buyer System

Suppose all of the $A_i$'s (i=1 to 7) are atomic remote procedure calls and the execution times for $A_1$ to $A_6$ are all 1. The execution time of $A_7$ is 1.2. For convenience, we denote the parallel block of the program as $C$. We have:

$$E_C = \frac{1}{5} \sum_{i=1}^{5} i \times Q_i^{1/5} = \frac{1}{126} \times (1 \times 1 + 2 \times 50 + 3 \times 50 + 4 \times 20 + 5 \times 5) = \frac{356}{126} \approx 2.8.$$  
That is, the execution time for $C$ is 2.8 time unit. So the evaluation time of this RPC program is 5.0 time units.

Now, if we assume the distributed system has only 3 hosts instead of 5, then

$$E_C = \frac{1}{3} \sum_{i=1}^{3} i \times Q_i^{1/5} = \frac{1}{21} \times (1 \times 0 + 2 \times 3 + 3 \times 9 + 4 \times 6 + 5 \times 3) = \frac{72}{21} = 3.4.$$
So the evaluation time of this RPC program will be 5.6 time units.

4.2. Extended Seller-Buyer System

Next we extend the above Seller-Buyer system by adding two wholesalers. That is, the Seller concurrently sends out the goods information to some Buyers and two wholesale-persons. These two persons then send the information to their own customers (Buyers) with their own comments which may influence these Buyers. All the bids return to the Seller. At last, the Seller will send the goods to the Buyer with the best offer. The RPC program structure corresponds to the flowgraph of figure 3 (we assume there are 3 hosts in this system). Here $A_1$ and $A_5$ are the same as the $A_1$ and $A_7$ of the last example, respectively. $A_2$ is the remote procedure of the first wholeseller, while $A_{22}$ and $A_{22}$ are its customers' remote procedure calls. $A_4$ is the RPC of another wholeseller and $A_{44}$ are the RPCs of the customers.

![Diagram of Extended Seller-Buyer System](image)

Fig. 3. Structure of the Extended Seller-Buyer System

Suppose the completion times of all the atoms are 1 unit except that $A_4$'s completion time is 1.2. To this time the derivation of a closed form solution to $Y_n$ for this system is intractable. Instead we shall indicate the use of algorithm (6) to compute the execution time for a particular map. At first, all the arcs are white; $CurrentTime = 0$, $s.CompletionTime = 0$; and arc $A_1$ is labelled red. Next, $A_1$ is coloured black and $CurrentTime = A_1.CompletionTime = 1$; $n_1.CompletionTime = 1$; and arcs $A_2$, $A_3$, $A_4$ are labelled blue. Also we suppose $A_2$ and $A_4$ are allocated on the host 1 and $A_3$ the host 2. Now we are at the second WHILE of the algorithm. We can select arc $A_2$ and colour it red, and $A_3.CompletionTime = 2$. Next we can select one of $A_2$ or $A_4$. Suppose we selected $A_4$. We colour it red and set $A_2.CompletionTime = 2$. There are no remaining blue arcs which can executes, so we go to the first WHILE and select one of the red arcs, say, $A_2$, and colour it black. Now $CurrentTime = 2$ and arcs $A_{21}$, $A_{22}$ are coloured blue. In this case arc $A_4$ can execute because no red arc is labelled with the same host (host 1). So we have $A_4.CompletionTime = 3$ because now $CurrentTime = 2$. In turn we have $n_2.CompletionTime = 2$, $n_3.CompletionTime = 3$, and $n_4.CompletionTime = 2$. Continuing this process we arrival node $f$ with $f.CompletionTime = 6.2$. This is the program execution time.

5. CONCLUSIONS

We have presented a performance evaluation model for RPC programs. Then we mentioned a non-deterministic algorithm for general case solution. After that, a closed-form solution for a special case is given and two examples are presented.

REFERENCES


357
ON THE MANAGEMENT OF REMOTE PROCEDURE CALL TRANSACTIONS

Wandei Zhou
Brian Molinari
Department of Computer Science
The Australian National University
ACT 2601, Australia
e-mail: zhou@anucs.anu.oz.au Phone: (06)249-3785

ABSTRACT

This paper describes the problems arising in managing remote procedure call (RPC) transactions. The structures and constructions of RPC-based systems have been widely discussed, while how to maintain the RPC transactions by using these existing structures and constructions remains unclear. This paper presents a model for maintaining the single and parallel RPC transactions. Some properties of the model are also described.

Key Words: Distributed Computing; Remote Procedure Call (RPC); Transaction Management.

1. Introduction

Transaction management is a well-established concept in database systems research. Two kinds of transactions are defined over an object system (a system with a collection of objects, and an object here can be a database file, an entry of the database file, or other proper things): the atomic transaction and the nonatomic transaction. An atomic transaction is defined as a sequence of operations which has the following two properties:\(^1\,^5\,^11\):

(i) Recoverability. The overall effect of a transaction is all-or-nothing: either all of the object states changed by the transaction remain in their states before the transaction, or all changed to their final states.

(ii) Indivisibility. The partial effects of a transaction is invisible to other transactions. That is, if the objects being modified by a transaction are observed over time by another transaction, then the latter transaction will either always observe the initial states before the former transaction, or the final states after the former transaction.

A nonatomic transaction is similar to an atomic transaction except that the second property is no longer held. Non-atomic transactions are needed when the duration of a transaction is long so that it is intolerable to wait for a transaction commits before the next transaction begins on the same data objects.\(^3\)

The needs for transaction-oriented RPC calls are obvious. For example, in our distributed calendar application\(^13\), a user usually issues a meeting which involves a group of people. Suppose that everybody in the group is a key person to the meeting, that is, if anybody cannot attend the meeting, then the meeting period must be re-arranged by the issuer. As we have known, the calendar database of those participants may locate on several different hosts. So, if anything goes wrong (such as the system failure of the related hosts or the time periods of some participants are already occupied and cannot be re-allocated) during the meeting arrangement call, which is a concurrent call of several RPCs, we need a method to rollback those calls that have been performed.

Existing RPC implementation or proposals usually do not deal with the transaction management of a RPC call or even a set of RPC calls because of the difficulties. Almost all the existing RPC
implementations use at-most-once as the calling semantics. That is, when a return message is obtained, the user can have confidence that the remote procedure has been executed exactly once. Otherwise, an exception will be raised and the user does not know if the RPC is executed or not. For example, if the RPC request message is lost, the remote procedure will not be executed; while if the reply message from the remote procedure is lost, then the remote procedure does executed. But in both cases the client can only know that something is wrong between the client and server from the raised exception. It cannot tell if the remote procedure is or is not executed.

Liskov and her colleagues considered the atomicity of RPC from the viewpoint of programming languages. Concepts such as guardians, actions, atomic data types, and promises are introduced to ensure the activity of some segments of a program. But a programmer has to incorporate into his program many "new" segments that deal with the atomicity of the RPC calls. In that case the programming is not so easy. They also do not consider the atomic transactions with server/client structure in usual RPC-oriented systems and applications.

We are not going to consider the full atomic RPC transaction within this paper. Instead, we only consider non-atomic transactions. So, from now on when we say transaction, we understand it means non-atomic transaction.

Two problems are to be dealt with in this paper:

- Single RPC transaction management: The transaction management of a single RPC call.
- Parallel RPC transaction management: The transaction management of a set of parallel RPC calls.

The remainder of this paper is organized as follows. Section 2 introduces the RPC transaction models used in our transaction management. Section 3 presents the design of a management system for single RPC transactions, and properties of the system are described. Based on that, Section 4 describes a system for managing a set of (parallel) RPC calls. Properties of the extended system are also described.

2. RPC Transaction Models

In this section we consider the RPC model from the viewpoint of a user process. Let

\[ P = \{ p \mid p \text{ is a remote procedure call} \} \quad \text{and} \quad A = \{ a \mid a \text{ is an argument list of a RPC call} \}. \]

After we made a RPC call, the call may return successfully or failed. We divide all errors of a call into two classes. The first class includes those errors by which we are definitely sure that the call is not performed. Whereas the second class includes those errors that we cannot tell whether the call is performed or not. So we abstract the type of a RPC call as a mapping

\[ c : P \times A \rightarrow \{ OK, FL, US \}. \]

**OK** means that everything is perfect during the RPC's execution and the RPC call is successful. **FL** means accessing failure. It means that the RPC is not executed. **US** means unknown state. That is, we do not know if the remote procedure has been executed or not.

We may have several strategies to deal with the case of receiving a **US** type. For example, a strategy might be:

- Query the system so that we are told if the RPC really happened.

  If it really happened, give us the result. Otherwise we can explicitly call it again.

  But the strategy is not applicable because the query may last intolerably long. For example, if the links between the server and the client are down, the query will never have a correct result until a link is recovered. So we adopt the following strategy:

  - We provide a rollback operation that will reverse the call if it happened and do nothing if it did not happen. The operation is defined as:

\[ r : P \times A \rightarrow \{ OK, FL, US \}. \]

Set \( \{ OK, FL, US \} \) has the similar meaning as above. If no confusion will be caused, we may simplify a RPC call \( c(p, a) \) into \( c(p) \), and a rollback call \( r(p, a) \) into \( r(p) \).
The rollback operation is managed by the system. When a rollback operation is issued, the user can go back to his own work immediately instead of waiting for the result. The system will guarantee that the RPC will be rolled back. We have the following assumptions (their detail explanation can be found in [13]):

(1). All RPC operations can be rolled back. And all RPC mappings are idempotent operations.
(2). The First-In-First-Serve (FIFS) strategy is used to order all operations of a server (except parallel operations).
(3). All orphaned calls2 are exterminated before a host recovers from crash to normal operation.
(4). When a RPC updates a data item, it locks the data item until the transaction the RPC belongs to finishes (commit or aborts). RPCs in other transactions that want to update the same data item will have FL returned.

Now we can describe our RPC transaction models. We define a single RPC transaction as $T = \{c(p, a)\}$. Where $c(p, a)$ is a RPC call. If there is no confusion, we also denote $T$ as $T = \{c(p)\}$. The semantics of a single RPC transaction is that after issuing the transaction, $c(p)$ will be executed if everything is all right, or will be rolled back if something is wrong.

We define a parallel RPC transaction as $T = \{c(p_1, a_1), \ldots, c(p_m, a_m)\}$. Where $c(p_i, a_i)$ (1 = 1, 2, ..., m) is a RPC call, and $p_i: F$, $a_i: A$. If there is no confusion, we also denote $T$ as $T = \{c(p), \ldots, c(p_m)\}$. The semantics of a parallel RPC transaction is that after issuing the transaction, all $c(p_i)$ of $T$ will be executed, or if any one of them fails, all executed RPCs will be rolled back. The execution of all $c(p_i)$ in $T$ is in parallel. Some parallel primitives can be built for parallel execution of remote procedures2. Sequential executed transaction can be easily established from the parallel model.

3. Single RPC Transaction Management

3.1. The Manager

Each host in the network has a RPC transaction manager (RM) executing on it. Let each host be assigned a unique number and $\{1, 2, \ldots, N\}$ is the set of host numbers, the set of RMs in the network can be expressed as $\{RM_1, RM_2, \ldots, RM_N\}$. A RM consists of a managing server and three tables which are stored in the stable storage2. These tables are:

**LET** Local Executed RPC Table. When a RPC is performed by a server in the host, it is reported to the RM and stored into LET. The function of LET table is to denote the executed RPCs and report them to the calling hosts. An entry of the table is defined as:

```c
typedef struct let {
    char *rpc;  /* name of the finished RPC call */
    int host_from;  /* client host number of the RPC call */
} LET;
```

**UST** Unknown State Table. When a RPC call finds the return is US, then the RPC call is recorded by the RM into the UST. Combining the local UST table with related LET entries of other RMs, we can decide which entries in the UST are executed and which are not. The definition of a UST entry is:

```c
typedef struct ust {
    char *rpc;  /* name of the unknown state RPC call */
    int host_to;  /* destination server host number */
    long time;  /* local timestamp when issuing the call */
} UST;
```

**NRT** Needed Rollback Table. If the RM decided that a RPC is to be rolled back, it is put into the NRT. A NRT table entry is defined as:

```c
typedef struct net {
    char *rpc;  /* name of the needed rollback RPC call */
} NET;
```
We denote the tables on host \( j \) as \( \text{LET}_j, \text{UST}_j, \) and \( \text{NRT}_j \), respectively.

### 3.2. Algorithms

The RM's algorithm is described in Algorithm 1, where \( \text{LET}, \text{UST}, \) and \( \text{NRT} \) tables and their counters are located in the stable storage. When the RM is first invoked, all these tables are set to empty and their counters are set to 0 by the initialization function. Constant \( \text{MAXENTRIES} \) is defined as the maximum number of entries in each of these tables. The RM forks into five "forever running" concurrent processes when it is invoked.

\begin{verbatim}
LET *let_table[MAXENTRIES]; /* LET table */
int let_ct; /* count of LET table */
UST *ust_table[MAXENTRIES]; /* UST table */
int ust_ct; /* count of UST table */
NRT *nrt_table[MAXENTRIES]; /* NRT table */
int nrt_ct; /* count of NRT table */

initialisation();

COBEGIN
    send_my_let(let_table); /* periodically send out entries in LET */
    rollback_nrt(nrt_table); /* periodically rollback entries in NRT */
    listen_extern_lets(ust_table, nrt_table); /* receive & process LETs of other RMs */
    manage_rpc_calls(ust_table); /* manage the RPC calls issued by its clients */
    listen_local_servers(let_table); /* put executed RPC names into LET */

COEND

The first function \( \text{send\_my\_let()} \) periodically groups all entries in the LET table according to their \( b.host\_from \) fields, and sends them out to that host. If the sending of a group of entries (called a \( \text{LET entry package} \)) is successful, those entries are deleted. The algorithm of the function is as follows:

\begin{verbatim}
send_my_let(let_table)
LET *let_table();
{
    int j, k;
    LET *grp_let[MAXENTRIES];
    while (TRUE) {
        j = the first host number;
        while (j != 0) {
            * loop until all host numbers are checked
            i loop over all let_table entries which have the same b.host_from field into grp_let table
            k is the returned number of entries in grp_let
            if (k != 0) {
                * send out the LET entry package grp_let if the sending is successful, delete these entries
                if (send_out(grp_let, k) == SUCCESS)
                    delete_let(grp_let, j, let_table);
            }
            j = next host number (if no next, j = 0);
        }
    }
}
\end{verbatim}

Function \( \text{delete\_let(grp_let, j, let_table)} \) deletes all the LET entries that belong to both \( \text{grp\_let} \) and \( \text{let\_table} \). We denote the LET entry package obtained in Algorithm 2 as \( \text{LET}_j \) if the local host is \( i \) and the remote host is \( j \).
The second function periodically rolls back all entries in the NRT table. If a rollback is successful, the entry is deleted. Next is the algorithm of the function:

```c
rollback_nrt(nrt_table)
NRT *nrt_table[];
{
    while (TRUE) {
        for all b ∈ NRT do {
            b = x(b); /* rollback it */
            if (b == OK) /* if rollback succeeds, delete b */
                delete b from NRT;
        }
    }
}
```

Algorithm 3

The third function is to listen to the sending of LET entry packages from other RMs and process them according to the local UST table. Suppose the local host is j and one of the remote hosts is k. If a RPC call issued by host j is performed by the destination host k, then the destination host k will finally send this message back through the LET entry package extracted from LET_i table (see Property 3.4 in Section 3.3 later). Now host j will check into its own UST_i table against the received LET_i tables entries. Suppose we now received the LET entry package LET_i from host k. If any entry is found in both LET_i and UST_i, that means the call returns US and it is actually executed by host k. So it is put into the NRT_j table for rolling back. If a call returns US but it is not executed by the destination host, there will be no such entry in the received LET_i package. In that case, we need to find the largest issuing time T of all RPCs executed by host k and issued by j, by joining the UST_i table and LET_i table together. Then we can delete those entries in UST_i which host_to field is k and the issuing time plus the maximum delay time is larger than the largest issuing time T found above. The function is described in Algorithm 4:

```c
listen_extern_lets(ust_table, nrt_table)
UST *ust_table[];
NRT *nrt_table[];
{
    LET *rcvd_let[MAXENTRIES], *b;
    UST *c;
    int k;
    long T;
    while (TRUE) {
        /* listen to the sending of LET entry packages. the process suspends until a sending comes */
        /* the first received package is put into rcvd_let */
        /* other sentings are suspended until the local host processes over the received package */
        /* k is the host number which sends the package */
        k = receiver(rcvd_let);
        /* processing the executed RPCs on host k put them into NRT if they appear in local UST */
        for all b ∈ rcvd_let {
            if (b.rpc = c.rpc where c ∈ UST) {
                NRT += b.rpc; /* here means insert */
                delete c from UST;
            }
        }
        /* processing other RPCs to host k and return US delete them from UST if they are not executed */
        /* T = largest time of executed RPC */
        T = max(c.t | c ∈ UST, b ∈ rcvd_let and b.rpc = c.rpc);
        for all c ∈ UST {
            if (c.t ≤ T + MAX_DELAY) and (c.host_to == k)
                delete c from UST;
        }
    }
}
```

Algorithm 4
delete c from UST; /* delete entries whose RPCs are not executed */
}
}
}

The fourth function of RM manages single RPC transaction calls of its clients. If any client program wants to make a RPC call, it calls the `manage_rpc_calls()` function and hands the call to the function. The function makes the RPC call. If the call is successful or not executed, it simply returns these messages to the client program. If the call returns `US`, then a UST entry is created and stored into the local UST table.

```
manage_rpc_calls(ust_table)
UST *ust_table[];
{
    int k;
    long t;
    UST *b;
    while (TRUE) {
        /* listen to local RPC call p the process suspends until a call p comes */
        /* if more calls come, they are queued up until the processing of the current call is over */
        listen_call(p);
        s = c(p); /* do the RPC call */
        k = the destination host number;
        t = the time when issuing p;
        switch {
            case (s == OK): /* The RPC executed and returns OK */
                break;
            case (s == FL): /* the RPC is not executed */
                break;
            case (s == US): /* the RPC may or may not executed */
                initialise b;
                b.rpc = p; b.host_to = k; b.time = t;
                UST += b;
        }
        tell the client the RPC returns s;
    }
}
```

The fifth function is to listen to the local servers and log their executed RPC names into the local LET table. So, each server will report its work if it has performed a RPC call. Two phase protocol can be used here to ensure that when a server reports its execution of a RPC, it will be executed even in the presence of failures. Next is the algorithm:

```
listen_local_servers(let_table)
LET *let_table[];
{
    char *p;
    int h;
    LET *b;
    while (TRUE) {
        /* listen to the report of executed RPC p, the process suspends until a report comes */
        /* if more reports come, they are queued until the processing of the current call is over */
        h = listen_server(p); /* h is the reported client host number */
        initialise b;
        b.rpc = p; b.host_from = h;
        LET += b;
    }
}
```
3.3. Properties

Before describing and proving properties of the model, we make the following assumptions. It is easy to know these assumptions are mostly realistic.

1. The life-cycle of system entities (hosts, links, and servers) is work $\rightarrow$ down $\rightarrow$ repair $\rightarrow$ start $\rightarrow$ work. Without losing generality, we assume that the work time and down time (include repair and restart time) are finite and denote them as $T_w$ and $T_d$, respectively.

2. The maximum delay time of a RPC call is $MAX\_DELAY$. That is, if a RPC call is issued at time $t$, then at time $t + MAX\_DELAY$, that call is either executed or will never be executed.

3. The average time of making a RPC call is relatively long. Particularly, it needs at least the same time as the average time that a RM sends out one of its LET entry packages, as well as at least the same as the average time of a rollback call.

4. The probability of a RPC call or a rollback call that returns OK is much larger than the probabilities that return FL or US. That is, in most cases a RPC call or a rollback call will be successful.

Following properties can be established for our single RPC transaction management model. We outline informal proofs here.

Property 3.1. If a RPC returns OK, it is executed and will not be rolled back.

Proof: By definition, the RPC is executed correctly when OK returns. Rollbacks are only performed in rollback state() algorithm according to entries in NRT table. While entries in NRT table come from UST table by the working of algorithm listen_exten_send(). According to algorithm manage_rpc_calls(), if a RPC call returns OK, it will not be put into the UST table, and so will not be put into the NRT table. So the property is true.

Property 3.2. If a RPC returns FL, it is not executed and will not be rolled back.

Proof: Similar as above.

Corollary 1. The time of sending out any entry in UST table is finite.

Proof: Algorithm 2 send_my_last() is responsible of sending out LET table entries. Because we grouped together all LET entries which have the same host, Secure before sending them out, so ideally we can view the LET table as a circular table of $N$ components, where $N$ is the number of hosts in the system. Each component contains a LET entry package. Now, the function of Algorithm 2 is to circularly send out all entries, one LET entry package each time. If the time needed to send out a LET entry package is 1 time unit, then the time needed to have a circular sending is $N$ units (we ignore the local processing time because they are very small compared to the communication time needed by the sending). That is, any entry of the LET table will be sent out in $N$ time units if no failures occur. As we have assumed the maximum down time for a host is $T_d$, that means the time for sending out the LET entry package of a failed host is at most $T_w + T_d$. So any LET entry will be sent out in finite time. From now on, we denote this time as SEND_LET_TIME.

Corollary 2. The size of any LET entry package is finite.

Proof: As we know, while we processing the LET table by using Algorithm 2, the RM will concurrently fill in LET by using Algorithm 6 listen_local_send()(). Let us assume the time needed to send out a LET entry package is 1 time unit and the average numbers of RPCs performed in a host in a time unit is $r$. The worst case is that all the RPCs are called from one host $i$. As Corollary 1 tells us that the time to send out any entry of LET is $N$ when no failures occur, so the maximum length of the LET entry package for host $i$ is $r \times N$. If host $i$ fails, or the links to the host $i$ fail, then no RPC calls from host $i$ can be successful. That means the length of any LET entry packages for host $i$ will not grow until the failures are recovered. Other cases are the same. So, the corollary holds.

Corollary 3. The time of rolling back all entries of NRT table is finite.

Proof: If we can prove that the speed of filling NRT table is less than the speed of deleting NRT table, the corollary will hold. Rollback and delete NRT entries is the responsibility of algorithm
rollback_rpc). While rolling back, the RM will concurrently fill in NRT by using function listenExtern_list() from UST table. UST table is filled in by function manage_rpc_calls() when RPCs return US. From assumption 4, only a small portion of RPCs need rollback, and assumption 3 tells us that the speed of rolling back is at most the same as the speed of RPC calls. So, if on average there are s RPCs performed in a time unit, then the filling speed to NRT table is s * p, where p is the probability that a RPC fails and p < 0.5. While the rollback speed is also s with s * q rollbacks return OK in a time unit, where q > 0.5. Because s * p < s * q, the corollary holds. We use ROLLBACK_NRT_TIME to denote this time.

Property 3.3. If a RPC returns US, it will eventually be rolled back if it is executed.

Proof: Suppose US = c(p), and the client is on host i, the server is on host k, and the timestamp is t. According to function manage_rpc_calls(), an entry will be put into UST table. We denote it as d, and d.rpc = p, d.host_to = k, and d.time = t. After a finite time period (at most SEND_TIMEOUT long by Corollary 1), host k will send all the names of performed RPCs (that were issued by host i) to host i. Suppose now all RPCs from host i to host k are processed before the sending. If p has been executed, then there is a b : B_j, such that b.rpc = d.rpc = a, where B_j is the set of all entries in LET_i with host name fields equal to i (that is, the LET entry package from host k to host i). In that case, according to function listenExtern_list(), p will be put into NRT table. After a while (at most ROLLBACK_NRT_TIME long according to Corollary 3), the RM_i will issue the rollback operation r(p) in function rollback_rpc(). If p is not executed, there will be no such b in B_j. So, the second segment of function listenExtern_list() will delete d from UST table, and no rollback operation is needed.

Now suppose p is performed just after k sent out package B_j. In that case, b will be put into LET_i but not in B_j. Because the RPC returns US, d is put into UST_i. In function listenExtern_list(), RM_i will find that d.time > T + MAX_DELAY because of assumption 2. So, d will remain in the UST table. Eventually, MAX_DELAY time will pass and when host k sends the B_j next time, b will be there and will be rolled back as above.

Because the round-time periods for function send_my_list() and function rollback_rpc() are finite, the MAX_DELAY time is finite, and the time between two neighbour works for system entities are finite, the above rollback operation will eventually take place.

Property 3.4. The LET, NRT, and UST tables will not grow indefinitely and any entry will be sent out or deleted eventually.

Proof: From Corollary 1 and 2, the finite host numbers in the system and the limited down time, it is easy to know that the assertion for LET table is true.

The UST table is filled in by function manage_rpc_calls(). It is easy to know that the average length of UST table is less than or equal to the average length of LET table. Because the UST contains only those RPC entries which return are US’s, while the LET contains the OK return entries. By assumption 4 the assertion holds. From the assumption that the down time of any system entity is definite, we know that all RPCs issued by a host will be acknowledged and checked in function listenExtern_list(). So, we can conclude that any entry in the table will be deleted eventually.

From Corollary 3, the speed difference between NRT filling and rolling back, and the limited down time, it is easy to know that the assertion for NRT table is true.

4. Parallel RPC Transaction Management

4.1. Algorithms

The RPC manager and algorithms used by RMs in the network when processing parallel RPC transactions is almost the same as in processing single RPC transaction calls except the function manage_rpc_calls() is extended to include the parallel RPC transaction processing.

The extended function at first listens to a parallel RPC transaction call from the local host and processes them concurrently. If all of the RPCs returns OK, or there are error returns but no RPCs are
performed in the parallel call, then it simply tells the client the result and exits (to listen to new parallel calls). If there are some error returns and there are some RPCs were performed in the parallel call, then a rollback operation for all those performed RPCs is issued and their results are collected. If all rollbacks return OK, then it tells the client that the transaction is failed but all performed RPCs are rolled back. If some of the rollback operations do not return OK, then they are put into the NRT table for further rolling back by the RM and the algorithm tells the client that the transaction failed and some rollbacks are to be performed by the RM. Next is the extended function:

```
manage_rpc_calls(ust_table)
UST ust_table();
|
UST *b;
while (TRUE) {
    /* listen to local RPC transaction calls. (suspends until a transaction call comes) */
    listen_call(p1, ... pn);
    CODEBEGIN /* concurrently execute the RPC calls within the transaction */
    reti = c(pi);
    ki = the destination host number,
    ti = the time when issuing c(pi), i = 1, 2, ..., m;
    CODEND;
    SUS = {p1 | reti = US};
    SFL = {p1 | reti = FL};
    SOK = {p1 | reti = OK};
    switch {
        case (SUS = SFL = O): /* all RPCs executed and returned OK */
            Algorithm 7
            tell the client the transaction returns OK;
            exit;
        case (SFL = O and SUS = O): /* error, but none RPCs are executed */
            ERR = FL; break;
        case (SUS = O): /* error, but some RPCs maybe executed */
            for each p1 in SUS
                b.rpc = p1; b.host_to = ki; b.time = ti;
                ust += b;
            ERR = US; break;
    }
    /* arrive here only when failures */
    if (SOK = O) /* no RPC is executed */
        tell client: transaction returns ERR and none needs rollback;
        exit;
    else {
        CODEBEGIN /* roll back */
        reti = r(pi), pi in SOK;
        CODEND;
        RU = {p1 | reti = US};
        RFL = {p1 | reti = FL};
        if (RU = RFL = O) /* all rollbacks are OK */
            tell client: transaction returns ERR and all rollbacks are done;
            exit;
        /* some rollbacks are not performed, log them into NRT */
```
NRT + p; p ∈ Ru ⋃ Rfu

tell client: transaction returns ERR and some rollbacks are not done;
exit;
| * else *
| * while *
}

4.2. Properties

It is evident that Property 3.4 of Section 3.3 holds for parallel RPC transactions. The following properties can be easily established:

Property 4.1. If a RPC transaction returns OK, all its RPCs are executed and will not be rolled back.
Proof: The only place for the system to return OK is when Sru = Srul = Ø. In that case, all RPC are executed correctly, and nothing is put into UST table. So, they will not be rolled back.

Property 4.2. If a RPC transaction returns FL, any executed RPCs will be rolled back.
Proof: The ERR is assigned to FL when Sru ≠ Ø and Sru = Ø. If Sru = Ø now, then the algorithm exits and no rollbacks are needed. Otherwise, the necessary rollbacks are performed. If all rollbacks return OK, the algorithm exists and all rollbacks are done. If any rollbacks fail, they are put into the NRT table and according to Property 3.4 of Section 3.3, they will be eventually rolled back.

Property 4.3. If a RPC transaction returns US, any executed RPCs will be rolled back.
Proof: The ERR is assigned to US when Sru ≠ Ø. In that case, all those unknown state RPCs are put into the UST table. According to Property 3.4 of Section 3.3, those entries will be put into NRT table and rolled back eventually. Other situations are similar as above.

5. Remarks

The design of a RPC transaction manager is described in this paper. After the introduction of the problem and the RPC transaction models, we used two steps to solve the RPC transaction management problem. At first, a system for managing transactions of single RPC calls is developed. Algorithm and properties of the system are described. Then, by extending the single RPC transaction management system, the parallel RPC transaction management system is described.

Almost all transaction management approaches use two-phase protocol. It is easy to modify our model into two-phase paradigm. In that case, the system will be able to deal with RPCs that are not capable of being rolled back. But the efficiency of two-phase model will be less than our model, mainly because the executability checking (checking if the RPC will return OK or not) will take much time, and the stable storage management in two-phase protocol is more complex than our model.

Our model is transparent to programmers. It can act as a run-time system within the programming environment. Programmers will not have too much burden to maintain the RPC transactions in their programs. They can use RPC transaction calls as usual RPC calls and the system will do all the job. We feel this is better than the language level implementation.

Several extensions of the model is possible. For example, one may want to explore the nested RPC transaction model, or extend the model to maintain the atomic RPC transactions.

References


A System for Managing Remote Procedure Call Transactions

Wanlei Zhou
School of Computing and Mathematics, Deakin University, Geelong, VIC 3217, Australia

Brian Molinari
Department of Computer Science, The Australian National University, Canberra, ACT 2601, Australia

This paper describes the design and implementation of a remote procedure call (RPC) transaction manager. The structures and constructions of RPC-based systems have been widely discussed, yet how to manage the RPC transactions by using these existing structures and constructions remains unclear. We have designed a system for managing RPC transactions based on our RPC transaction model. The system is designed at first to manage single RPC transactions. It is then expanded to manage parallel RPC transactions as well. Some properties of the system are described. We also present some descriptions of the preliminary implementation of our RPC transaction manager.

1. BACKGROUND AND RELATED WORK

Transaction management is a well established concept in database system research. A transaction is defined as a sequence of operations over an object system (a system with an associated collection of objects, where an object can be a database file, an entry of a database file, or a model of a real-world entity such as printers or actuators of a control system). A transaction is indicated by the code fragment:

```plaintext
BEGIN_TRANSACTION
Operation 1;
Operation 2;
...
Operation n;
END_TRANSACTION
```

This sequence of operations must be performed in such a way that either all of them execute or none of them do. More specifically, a transaction has the following properties (Cary, 1981; Liskov and Sheifler, 1983; Couloris and Dollimore, 1988; Zhou and Lim, 1992):

1. **Recoverability.** The overall effect of a transaction is all-or-nothing: either all of the objects addressed by the transaction remain in their states before the transaction, or all are changed to their final states.

2. **Indivisibility.** The partial effects of a transaction are invisible to other transactions. That is, if the objects being modified by a transaction are observed over time by another transaction, then the latter transaction will always observe the initial states before the former transaction or the final states after the former transaction.

Because of these characteristics, transactions are used to provide reliable computing systems and a mechanism that simplifies the understanding and reasoning of programs.

Applications running on any distributed system typically have a number of requirements, including reliability, distribution, reconfiguration, concurrency, and consistency. Transactions are used to achieve most of these requirements by ensuring reliability and consistency in the presence of concurrency and failures. Besides this, transactions also help the programmer to organize programs for distributed systems by allowing them to view a complex piece of code as if it is run atomically.

The motivation for creating a transaction processing manager can be found in Bernstein (1990). In that paper, Bernstein also describes the general architecture model of a transaction processing monitor and describes in detail some of the components of...
the model. Gray and Reuter’s book (Gray and Reuter, 1993) provides more details on transaction processing.

There are many ways of invoking remote services in a distributed system, and the remote procedure call (RPC) mechanism is one of the preferred methods. This is because programs that use procedures are easier to understand and reason about than those that explicitly send and receive messages. The RPC has the same semantics as an ordinary local procedure call, except that the called procedure is executed in a different process and usually a different computer from that of the caller. The use of RPC facilitates the building of distributed programs, removing concern for the communication mechanisms from the programs that use remote procedures and leaving only the fundamental difficulties of building distributed systems such as synchronization and independent failure of components.

Currently, most of the existing RPC systems do not support transaction management because of the difficulties involved (Nelson, 1981). For example, the policy and expense of making each remote call or even a group of remote procedure calls atomic is too great a burden for many client programs. Also, remote procedures and atomicity are basically independent notions that require more investigation and experience before they are tied together. Most of the existing RPC implementations use at-most-once (Coulouris and Dollimore, 1988) as the calling semantics. That is, when a return message is obtained, the user can be confident that the remote procedure has been executed exactly once. Otherwise, an exception will be raised and the user does not know if the RPC has executed or not. For example, if the RPC request message is lost, the remote procedure will not be executed; while if the reply message from the remote procedure is lost, then the remote procedure will have been executed. But in both cases, the client can only know that something is wrong between the client and server from the raised exception. It cannot tell if the remote procedure has or has not executed.

In order to achieve a more stable distributed system, transaction-oriented RPCs are needed. Firstly, transaction-oriented RPCs provide a reliable mechanism for building stable distributed systems (because of the recoverability and the indivisibility). Secondly, transaction-oriented RPCs can simplify the structure of a distributed program and make the program easier to understand. Thirdly, by using transaction-oriented RPCs, we can greatly reduce the burden for many clients of distributed programs—we only need to specify the transactions in our programs. The transaction system will manage the atomicity for us. That is, we need to manage an RPC or even a set of RPCs as a transaction. Besides incorporating the transaction techniques and the RPC facility, management of these RPC transactions and appropriate security techniques are also needed to support a rich base for constructing reliable software systems. These give rise to the idea of an RPC transaction manager.

A general RPC transaction manager is responsible for scheduling, for system recovery, and for authorization. It allows collections of actions to be grouped to form atomic transactions. It ensures either that a transaction commits and its effects survive forever or that the transaction aborts and its effects are forgotten. The transaction manager insulates the application programmer from the problems of failures and concurrency. In addition, the transaction manager enforces restrictions on who may run each transaction type. In essence, the general RPC transaction manager extends the base operating system to support RPC transaction management concepts.

The Argus system (Liskov, 1988) developed by Liskov and her colleagues considers the atomicity of RPC from the viewpoint of programming languages. Concepts such as guardsians, actions, atomic data types, and promises (Liskov and Sheifer, 1983; Weihl and Liskov, 1985; Liskov and Shirira, 1983) are introduced to ensure the atomicity of some segments of a program. Argus allows computations (including remote procedure calls) to run as atomic transactions to solve the problems of concurrency and failures in a distributed computing environment. A user can also define some atomic objects, such as atomic arrays and atomic records, to provide the additional support needed for atomicity. All user fault tolerance requirements must be specified in the Argus language. That is, a programmer has to incorporate into his program many “new” segments that deal with the atomicity of the RPCs.

The atomic RPC system implemented on ZMOB (Lin and Gannon, 1985) uses sequence numbers and calling paths to control the concurrency and atomicity and uses checkpointing to maintain the ability to recover from failures. Users do not have to provide synchronization and recovery themselves; they only need to specify if atomicity is desired. This frees them from managing much complexity. It can only deal with single atomic RPCs.

In our work, we will concentrate on the management and fault tolerance of simple RPC transac-
tions. Two problems are dealt with in this paper:

- **Single RPC transaction management:** The transaction management of a single RPC.
- **Parallel RPC transaction management:** The transaction management of a set of parallel RPCs.

All these problems are considered under the server/client paradigm and are transparent to users. The purpose of the design is to provide programmers with the transaction management facility in a similar fashion as providing concurrent processing facilities or memory management facilities in many operating systems.

The remainder of the paper is organized as follows. Section 2 introduces the RPC model used in our transaction management. Section 3 presents the design of a management system for single RPC transactions, and properties of the system are described. Based on that, Section 4 describes a system for managing a set of (parallel) RPCs. Properties of the extended system are also described. Section 5 describes the preliminary implementation of our RPC transaction manager.

### 2. RPC TRANSACTION MODELS

#### 2.1 The RPC Model

In this section we consider the RPC model from the viewpoint of a user process. Let

\[ P = \{ p | p \text{ is a remote procedure provided by the system} \} \]

A remote procedure can be located at any host in the network. Let

\[ A = \{ a | a \text{ is an argument list of an RPC} \} \]

After we make an RPC, the call may return successfully or fail. There may be several reasons for the failure of a call. For example, in Apollo/HP's Network Computing System (NCS) (Zahn et al., 1990), one often sets up an exception segment before the real RPC to catch several errors such as "communication error" and "server error." There are also other errors that a user does not want to catch, but they may happen during an RPC. With some error conditions, it is clear to the client that the procedure was not executed, while with other error conditions, it is not clear if the procedure was executed or not. For example, by having a "server error," we are sure that the RPC is not performed. If we have a "communication error," however, the client will not be able to tell if the call is performed or not because we do not know whether the error happened before or after the calling request arrived at the destination host.

We define the effects of an RPC as the processing of a data item of the object system. Hence, we can abstract an RPC as a mapping of the following type

\[ c: P \times A \rightarrow \{ OK, FL, US \} \]

Here the values of the target set have the following meaning:

- **OK** This means that no failure occurred during the RPCs execution. By \( c(p, a) = OK \), where \( p \in P \) is an RPC and \( a \in A \) is the argument list of the call, we mean that the RPC call was successful (i.e., the remote procedure performed the job).

- **FL** This means accessing failure. By \( c(p, a) = FL \), we mean that the destination server (the server that exports the remote procedure \( p \)) could not perform the job because, for example, the arguments between the client and server do not match, the versions are different, or the object managed by the server is not accessible by the client. This means that the RPC has not executed.

- **US** This means unknown state. By \( c(p, a) = US \), we mean that the client cannot tell if the RPC has or has not been executed because, for example, the destination host (the host that the server exporting procedure \( p \) resides on) is down, or the links between the client and server are down (that is, the client host and the server host belong to two partitioned sub-networks). In that case, maybe the RPC request is lost, or maybe the return message is lost, so we do not know if the remote procedure has been executed or not.

We may have several strategies to deal with the case of receiving a US value. For example, a strategy might be:

- Query the system so that we are told if the RPC really happened.

- If it really happened, give us the result; otherwise, we can explicitly call it again.

But the strategy is not applicable because the query may last intolerably long. For example, if the links between the server and the client are down, the query will never have a correct result until a link is recovered. So we adopt the following strategy:

- We provide a rollback operation that will reverse the call if it happened and do nothing if it did not happen. The operation has the following type

\[ r: P \times A \rightarrow \{ OK, FL, US \} \]
The target set \( \{ \text{OK}, \text{FL}, \text{US} \} \) has a similar meaning as above. We assume that two applications of \( r \) have the same effect as one application. In mathematical terms, all \( r \) mappings are idempotent operations. This is reasonable because if an RPC has been rolled back, then any subsequent rollback commands have no effect. For example, if an RPC is to "reserve the required seat for customer X if it is free" on a flight reservation system, then the rollback operation may be "clean the reservation of customer X." If the reservation for customer X has been cleared, the rollback operation can simply be ignored. So, here \( \text{OK} \) means that the RPC has been rolled back, or it has been rolled back before by other rollback operations, or even the RPC is not performed. In order to have confidence that an RPC is really rolled back, one may keep using the \( r \) operation until an \( \text{OK} \) is returned.

Of course, some operations cannot be rolled back. For example, suppose the effect of an RPC is to print a check. When the rollback request comes, if the printing task is still in the printer queue, it is easy to do the rollback; but if the printing task has been performed, the computer system can do nothing about it. In our following discussion, we consider only RPC operations that can be rolled back.

If no confusion will be caused, we may simplify an RPC \( c(p,a) \) into \( c(p) \), and a rollback call \( r(p,a) \) into \( r(p) \).

The rollback operation will be managed by our transaction management system. When an RPC is returned, irrespective of whether it is successful or not, the user program can go back to its own work immediately instead of waiting for the rollback operation result if the call fails. If the call does fail, the system will guarantee that the RPC will be rolled back.

Now consider that there is only one user process, and an RPC \( c(p,a) \) is to be used to change the state of a data item \( D \) from \( S_0 \) to \( S_1 \). If \( c(p,a) = \text{OK} \), then \( D \) is in \( S_1 \). If we want to roll back, then \( r(p,a) = \text{OK} \) will guarantee that the RPC has been rolled back, that is, \( D \) is now in \( S_0 \). If \( c(p,a) = \text{FL} \), then \( D \) is still in \( S_0 \). If we also do the rollback operation, then \( r(p,a) = \text{OK} \) means that there is no effect caused by the operation. That is, \( D \) remains in \( S_0 \). When \( c(p,a) = \text{US} \), the user does not know what is the current state of \( D \). If it is in \( S_0 \), then \( r(p,a) = \text{OK} \) has no effect on \( D \), and \( D \) still remains in \( S_0 \); if it is in \( S_1 \), then \( r(p,a) = \text{OK} \) will make \( D \) back to \( S_0 \). Figure 1 illustrates the above discussion.

There may be other possibilities when \( c(p,a) = \text{US} \). As a first possibility, the RPC may be delayed by the system. For example, in Figure 2A, the destination computer (B) crashes after it receives the RPC request and before executing it. In that case, the RPC call will resume after host B recovers. But for the calling process, it will receive a US message (any reply from the server after that will be ignored) and a rollback operation will be issued. Of course, the rollback operation cannot succeed before host B recovers. But after host B recovers, both the RPC and the rollback operations will be carried out, and usually, we cannot guarantee which one will be performed first. Now, if the rollback operation performs first and \( r(p,a) = \text{OK} \), the user may think that the RPC has been rolled back, and \( D \) is in \( S_0 \). But after the rollback operation (which leaves \( D \) in \( S_0 \)), the original RPC request performs and makes \( D \) change from \( S_0 \) to \( S_1 \). One possible solution to this problem is to have the recovery process of each host kill all incomplete RPC operations before it returns to normal execution. In that case, the above original RPC request will be killed during recovering and the user can have the confidence that \( D \) will remain in \( S_0 \) after the rollback operation.

The second possibility is that the calling host (A) may crash just after its process issued the RPC \( c(p,a) \), as indicated in Figure 2B. Then, after host A recovers and the process restarts, it will do \( c(p,a) \) again without knowing that there is already a same

\[ \text{Figure 1. RPC and its rollback operation.} \]

\[ \text{Figure 2. Some possibilities when return is US.} \]
A System for Managing Remote Procedure Call Transactions

J. SYSTEMS SOFTWARE
1996; 34:133–149

Figure 3. Semantics of the single RPC transaction.

2.3 Parallel RPC Transaction Model

We define a parallel RPC transaction as $T = \{c(p_1,d_1), \ldots, c_m(p_m, d_m)\}$, where $c_i(p_i, d_i)$ is an RPC, and $p_i \in P, d_i \in A$. If there is no confusion, we also denote $T$ as $T = \{c(p_1), \ldots, c_m(p_m)\}$. The semantics of a parallel RPC transaction is that after issuing the transaction, all $c_i$ of $T$ will be executed, or if any one of them fails, all executed RPCs will be rolled back. The execution of all $c_i$ in $A$ in parallel and partial effects are allowed. Some parallel primitives can be built for parallel execution of remote procedures (Zhou and Molinari, 1990). Sequential executed transaction can be easily established from the parallel model. Figure 4 indicates the semantics of a parallel RPC transaction. In this figure $d_i (i = 1, \ldots, m)$ are data items processed by the transaction.

2.4 Accessing Servers

In the client/server model, server processes manage objects and client processes access these objects by

Figure 4. Semantics of the parallel RPC transaction.
using the remote procedures exported by servers (Sinha, 1992). Each server can then be viewed as a
monitor (Hoare, 1974) and the object(s) it manages
can be viewed as a shared resource. The server
enforces the following rules:

1. The shared objects of the server can be accessed
only by the remote procedures of the server.
2. Only one client at a time can succeed in changing
the state of an object of the server. If a client is
already using the object, other clients that want
to use it are suspended until the object is free.
3. The remote procedures may be called by any
client.

These rules guarantee mutual exclusion between a
set of client processes that use the shared objects in
a server. Usually, a server and all its objects are
located at the same host.

Another observation is that a remote procedure is
far more complex than simple read or write opera-
tions used to model database-oriented transactions.
An RPC, compared with read or write operations,
has a higher level of abstraction and has other
features (such as communications) that are not
emphasized in the database concurrency control tech-
niques. Take our earlier flight reservation RPC as
an example. The RPC may involve reading the
reservation data base, checking the required state and
requirements, writing back to the data base if the
customer’s requirements are satisfied, as well as
communications between the client process and the
server process. So, the data base concurrency con-

3. SINGLE RPC TRANSACTION
MANAGEMENT

3.1 The Manager
To provide single-RPC transaction management
each host maintains an RPC manager (RM) with a
structure indicated by Figure 5.

We consider a distributed system of N hosts.
Without loss of generality, each host has a unique
identifier in the set \{1, 2, ..., N\}. Each host has a
local time clock. An RM consists of a managing
server and three tables which are stored in the
stable storage (Lampson, 1981). Information stored
in these tables will not be affected by system fail-
ures. These tables are:

\textit{LET Local Executed RPC Table}. When a server in a
host executes an RPC (for a remote client) it is
reported to the RM on that host and is stored in
the \textit{LET}. An entry in \textit{LET} has two fields. If
\( b_i \in \text{LET} \), then \( b_i.rpc = a \), where \( a \) is the name of
the finished RPC. It is kept unique for each call.
And \( b_i.host..from = h \), where \( h \) is the number of
the client’s host. The function of the \textit{LET} table is
to denote the executed RPCs so they can be
reported to the calling hosts. An entry of the table
is defined as:

\begin{verbatim}
typedef struct let {
  char *rpc; /* name of the finished RPC */
  int host_from; /* client host number of the RPC */
} LET;
\end{verbatim}

\textit{UST Unknown State Table}. When a client in a host
has an RPC return with U5 value, it is reported to
the RM on that host and is stored in the \textit{UST}. It
is impossible at this moment for the RM to tell if
the RPC has been performed or not by the server
host. There are three fields for each entry in the
\textit{UST}. If \( b_i \in \text{UST} \), then \( b_i.rpc = a \) (\( a \) is the unique

\begin{figure}[h]
\centering
\includegraphics[width=\textwidth]{rpc_manager.png}
\caption{Structure of the RPC Manager.}
\end{figure}
name of the RPC; \( b_{\text{host_to}} = h \) (\( h \) is the number of the destination/server host); and \( b_{\text{time}} = t \) (\( t \) is the time of issuing the RPC call and is stamped by the caller's host). Combining the local UST table with related LET entries of other RM's, we can decide which entries in the UST have been executed and which have not. The definition of a UST entry is:

```c
typedef struct ust {
    /* name of the unknown state RPC */
    char *rpc;
    /* destination/server host number */
    int host_to;
    /* local timestamp when issuing the call */
    long time;
} UST;
```

**NRT Needed Rollback Table.** If the RM determined that an RPC is to be rolled back, it is put into the NRT. Rollback commands for all entries in the table are issued periodically by the RM. Each NRT entry has one field. If \( b \in \text{NRT} \), then \( b_{\text{rpc}} = a \) (\( a \) is the unique name of the RPC which is to be rolled back). An NRT table entry is defined as:

```c
typedef struct nrt {
    /* name of the needed rollback RPC */
    char *rpc;
} NRT;
```

We denote the tables on host \( j \) as \( \text{LET}_j \), \( \text{UST}_j \), and \( \text{NRT}_j \), respectively.

### 3.2 Algorithms

Now we describe the behavior of each RM in the distributed system.

**Algorithm 1.**

\[
\text{LET } *\text{let_table}[	ext{MAXENTRIES}]; /* \text{LET table} */
\]

\[
\text{int let_ct}; /* \text{count of LET table} */
\]

\[
\text{UST } *\text{ust_table}[	ext{MAXENTRIES}]; /* \text{UST table} */
\]

\[
\text{int ust_ct}; /* \text{count of UST table} */
\]

\[
\text{NRT } *\text{nrt_table}[	ext{MAXENTRIES}]; /* \text{NRT table} */
\]

\[
\text{int nrt_ct}; /* \text{count of NRT table} */
\]

\[
\text{Initialization():}
\]

\[
\text{BEGIN}
\]

```c
/* periodically send out entries in LET 
send_my_let(let_table); */
/* periodically roll back entries in NRT 
rollback_nrt(nrt_table); */
/* receive LET entries from other RM's and process them 
listenExtern_lets(ust_table, nrt_table); */
/* manage the RPCs issued by its clients 
manage_rpc_calls(ust_table); */
/* listen to local servers and put executed RPC names into LET */
listen_local_servers(let_table);
```

\[
\text{END}
\]
The first function `send_my_let()` periodically groups all entries in the LET table, according to their `b.host_from` fields and sends them out to that host. If the sending of a group of entries (called an LET entry package) is successful, those entries are deleted. If there is no entry for host `j`, an empty entry package is sent to host `j` instead. The empty entry package is used to tell host `j` that the communications between the two hosts are normal. This message is then used by the Algorithm 4 `listenExternLets()`.

The algorithm of the first function is as follows:

```
Algorithm 2.
send_my_let(let_table)
LET *grp_let[MAXENTRIES];

int j;

LET *grp_let[MAXENTRIES];
while (TRUE) {
    j = the first host number (1);
    /* loop until all host numbers are checked */
    while (j != 0) {
        /* group all let_table entries which have the same
           b.host_from = j field into grp_let table */
        group(grp_let, j, let_table);
        /* send out the LET entry package grp_let */
        /* if the sending is successful, delete these entries */
        if (send_out(grp_let) == SUCCESS)
            delete_let(grp_let, j, let_table);
    }
    j = next host number (j++; if j>N then j = 0);
}
```

If the sending of the LET entry package is successful, the function

```
delete_let(grp_let, j, let_table)
```

deletes all the LET entries that belong to both `grp_let` and `let_table`. Otherwise, no deletion takes place and this LET entry package will be sent again in the next iteration. If the local host is `i`, and the remote host is `j`, then we denote the LET entry package obtained in Algorithm 2 as `LETj`.

The second function periodically rolls back all entries in the NRT table. If a rollback is successful, the entry is deleted. The algorithm follows:

```
Algorithm 3.
rollback_nrt(nrt_table)
NRT *nrt_table[];

{ while (TRUE) {
    for all b in NRT /* roll it back */
        s = x(b);
    /* if rollback succeeds, delete b */
    if (s == OK)
        delete b from NRT;
}
}
```

The third function receives LET entry packages from the other RM servers and processes them according to the local UST table. Suppose the local host is `j` and one of the remote hosts is `k`. If an RPC issued by host `j` is performed by the destination host `k`, then the destination host `k` will finally send this message back through the LET entry package extracted from `LETk` (see Property 3.4 in Section 3.3 later). Now host `j` will check into its own `USTj` table against the received `LETk` tables entries. Suppose we have now received the LET entry package `LETk` from host `k`. If any entry is found in both `LETj` and `USTj`, that means the call returned `US` and it was actually executed by host `k`, so it is put into the `NRTj` table for rolling back. If a call returned `US` but was not executed by the destination host, there will be no such entry in the received `LETk` package (or it is an empty package). In that case, we need to find out the `USTj` entries which are “old” enough and delete them. In fact, we can delete those entries in `USTj` which `host_to` field is `k` (e.g. `host_to` field is `k`), and the issuing time plus the maximum lifetime (c.t + MAX_LIVE) is less than (earlier than) the current time `T` of the local host. Because `c.t` is a time stamp of the local host (see Algorithm 5), `MAX_LIVE` is a constant, and `T` is the current time of the local host, so `c.t` and `T` can be compared. The function is described in Algorithm 4:
Algorithm 4.

```c
listen_exten_lets(ust_table, nrt_table)
UST *ust_table[];
NRT *nrt_table[];
{
    LET *rcvd_let[MAXENTRIES], 'b;
    UST *c;
    int k;
    long T;
    while (TRUE) {
        /* receive LET entry packages */
        /* the process suspends until a package arrives */
        /* the first received package is put into rcvd_let */
        /* other messages are blocked until the local host processes over the received package */
        /* k is the host number which sends the package */
        k = received(rcvd_let);
        /* processing the executed RPCs on host k */
        /* put them into NRT if they appear in local UST */
        for all b in rcvd_let {
            if (b.rpc = c.rpc where c in UST) {
                NRT += b.rpc;
                delete c from UST;
            }
        }
        /* processing other RPCs to host k and return UST */
        /* delete them from UST if they are too 'old' */
        /* (not executed) */
        T = current time of my host;
        for all c in UST {
            if (c.t + MAX_LIVE <= T) and (c.host |-c <= k)
                /* delete entries whose RPCs are not executed */
                delete c from UST;
        }
    }
}
```

Here MAX_LIVE is the maximum lifetime of an RPC. That is, if an RPC is issued at time t and there are no disconnections between the client and the server, then at time t + MAX_LIVE there is no possibility that the RPC will execute. The RPC has either executed or will never be executed by the system after time t + MAX_LIVE (refer to the assumptions in Section 3.3).

The fourth function of an RM manages single

Algorithm 5.

```c
manage_rpc_calls(ust_table)
UST *ust_table[];
{
    int k;
    long t;
    UST *b;
    while (TRUE) {
        /* receive local RPC p */
        /* the process suspends until a call p comes */
        /* if more calls come, they are queued up until */
        /* the processing of the current call is over */
```
listen_call(p);
    /* do the RPC */
    s = c(p);
    k = the destination host number;
    t = the time of the local host when issuing p;
    switch {
      case (s == OK): /* The RPC executed and returns OK. */
        break;
      case (s == FL): /* the RPC is not executed */
        break;
      case (s == US): /* the RPC may or may not executed */
        initialize b;
        b.rpc = p; b.host_to = k; b.time = t;
        UST += b;
    }
    tell the client the RPC returns s;
}

The fifth function is to listen to the local servers and log their executed RPC names into the local LET table. Each server reports its work if it has performed an RPC. Because the execution of the

Algorithm 6.

listen_local_servers(let_table)
LET *let_table[];
{
    char *p;
    int h;
    LET *b;
    while (TRUE) {
        /* receive the report of executed RPC */
        /* if more reports come, they are queued until */
        /* the processing of the current call is over */
        h = listen_server(p);
        /* p is the reported RPC */
        /* h is the reported client host number */
        initialize b;
        b.rpc = p; b.host_from = h;
        LET += b;
    }
}

3.3 Properties

Before describing and proving properties of the model, we make the following assumptions. It is easy to see that these assumptions are mostly realistic.

1. The life-cycle of the system entities (hosts, links, and servers) is

    loop
        work;
        crash;
        repair and restart;
    end loop

Without loss of generality, we assume that the work time and down time (including crash, repair, and restart time) are finite and are denoted as \( T_w \) and \( T_d \), respectively. Notice that the \( T_d \) is defined as the time interval during which a host does not function correctly. This includes crash, repair, and restart time of system entities (hosts, links, and servers) that affect the particular host.

2. The maximum lifetime of an RPC is \texttt{MAX\_LIVE}. That is, if an RPC is issued at time \( t \) and there are no disconnections between the client and the server, then at time \( t + \texttt{MAX\_LIVE} \), there is no
A System for Managing Remote Procedure Call Transactions

possibility that the RPC will execute. The RPC has either executed or will never be executed by the system after time $t + \text{MAX\_LIVE}$. This can be guaranteed partially by the orphan extermination algorithms and other methods used during system recovery (described in Section 2.1).

3. Sending (receiving) an LET entry package and making a rollback operation are all implemented by remote procedure calls. Hence, we assume that their average execution time is the same as an ordinary RPC.

4. The probability that an RPC returns OK is much larger than the probability that it returns FL or US. If we denote by $p$ the probability that an RPC or a rollback call returns OK, and denote by $q$ the probability of the other two returns, then we have $p + q = 1$ and $p > 0.5$. That is, an RPC or a rollback call will be successful in most cases if we view them over a sufficiently long period.

The following properties can be established for our atomic RPC management model. We outline informal proofs.

Property 3.1. If an RPC returns OK, it has been executed and will not be rolled back.

Proof. By definition, the RPC is executed correctly when OK returns. Rollbacks are only performed in the rollback_nrt() algorithm according to entries in the NRT table. In turn, entries in the NRT table come from the UST table by the working of algorithm listen_extern_lets(). According to algorithm manage_rpc_calls(), if an RPC returns OK, it will not be put into the UST table and thus, will not be put into the NRT table. So the property is true.

Property 3.2. If an RPC returns FL, it has not been executed and will not be rolled back.

Proof. Similar to the proof of Property 3.1.

Corollary 1. Any entry of the LET resides in the LET for a finite time.

Proof. The function send_my_let() is responsible for sending out and deleting LET table entries. Because we grouped together all LET entries which have the same host_from field before sending them out, we can view the LET table as having the structure indicated in Figure 6. Here $N$ denotes the number of hosts in the system. Now, Algorithm 2 uses a loop to send out entries, with one LET entry package each iteration. If the time needed to send out an LET entry package is one time unit, then the time needed to complete the loop has an upper bound of $N$ units (we ignore the local processing time because it is very small compared with the communication time). That is, if no failures occur, the upper bound of sending out of any entry of the LET table is $N$ time units. As we have assumed that the maximum down time for a host is $T_d$, that means the upper bound for sending out the LET entry package of a failed host is $T_d + N$. So, any LET entry will be sent out in finite time. From now on, we denote this upper bond as SEND\_LET\_TIME.

Corollary 2. The size of any LET entry package is finite.

Proof. Algorithm 6 writes entries to the LET structure, while Algorithm 2 deletes them. Let us assume that the time needed to send out an LET entry package is one time unit and that the average numbers of RPCs performed by a host in a time unit is $s$. The worst case is that all the RPCs are called from one host $i$. Corollary 1 tells us that the upper bound of sending out any entry of LET is $N$ when no failures occur, so the maximum length of the LET entry package for host $i$ is $s \times N$. If host $i$ fails, or the links to the host $i$ fail, then no RPCs from host $i$ can be successful. That means the length of any LET entry packages for host $i$ will not grow until the failures are recovered. Other cases are the same. So, the corollary holds.

Corollary 3. All RPCs that need to be rolled back will be rolled back within finite time.

Proof. According to our model, any RPCs that are to be rolled back are stored in the NRT. The system then is responsible of rolling them back by using Algorithm 3. We prove this corollary by two steps. At first we prove that the NRT will not grow indefinitely, then we prove that any entry in the NRT will be deleted (rolled back) within finite time.

Rollback and the deletion of NRT entries is the responsibility of function rollback_nrt(). At the same time, the RM server writes entries to NRT from UST using function listen_
extern lets(). In turn, the UST is written to by function manage_rpcs() when RPCs return US. From Assumption 4, only a small portion of RPCs need the rollback operation. If on average there are s RPCs performed in a time unit, then the NRT is filled at a rate of $s \times q$, where $q$ is the probability that an RPC fails and, according to Assumption 4, $q < 0.5$. In turn, on average there can be $s$ RPCs rolled back per time unit because of Assumption (3). That means there can be $s \times p$ rollbacks returning OK in a time unit, where $p > 0.5$. Because $s \times q$ is the speed of filling the NRT, and $s \times p$ is the speed of deleting NRT, and $s \times q < s \times p$, the NRT will never grow indefinitely (note that the $p$ and $q$ include both cases that the system is normal and abnormal, so we do not need to consider these cases separately).

If no failures occur, on each iteration the rollback operation $r(b)$ of function rollback_nrt() will succeed and $b$ will be deleted from the NRT. As we have proved that the NRT has a finite size, the for loop of function rollback_nrt() will finally go through all entries of the NRT. We may denote the upper bound time of this traversal as $Q$. Now suppose host $i$ is down, then any rollback operations to host $i$ will not return OK. As we have assumed that the maximum down time is $T_d$, it is evident that the upper bound time for these rollback operations being performed is $Q + T_d$. We use ROLLOBACK_NRT_TIME to denote this upper bound.

**Property 3.3.** If an RPC returns US, it will eventually be rolled back if it was executed.

**Proof.** Consider a remote procedure call $c(p)$, made from a client on host $i$ to a server on host $k$, and suppose that it returns US. According to function manage_rpcs(), an entry will be put into the UST table. We denote it as $d$, and $d.rpc = p$, $d.host_to = k$, and $d.time = t$. After a finite time period (at most SEND_LET_TIME by Corollary 1) host $k$ will send all the names of the performed RPCs (that were issued by host $i$) to host $i$. Suppose now that all RPCs from host $i$ to host $k$ are processed before this return. If $p$ has been executed, then there is a $b \in LET_i$, such that $b.rpc = p$, where $LET_i$ is the set of all entries in LET_i with host name fields equal to $i$ (that is, the LET entry package from host $k$ to host $i$). In that case, according to function listen_extern_lets(), $p$ will be put into the NRT table. After an interval (at most ROLLBACK_NRT_TIME according to Corollary 3), the RM will issue the rollback operation $r(p)$ in function rollback_nrt(). If $p$ is not executed, there will be no such $b$ in LET_i or LET_i will be empty. So, the second segment of function listen_extern_lets() will delete $d$ from UST table if $p$ has exceeded its maximum live time. Then no rollback operation is needed.

Now suppose $p$ is performed just after $k$ sent out package $LET_i$. In that case, $b$ will be put into $LET_i$ but not in $LET_i$. Because the RPC returns US, $d$ is put into $UST_i$. In function listen_extern_lets(), $RM_i$ will find that $d.t + MAX_LIVE > T$ because of assumption 2. So, $d$ will remain in the $UST_i$ table. Eventually, MAX_LIVE time will pass and when host $k$ sends the $LET_i$ next time, $b$ will be there and will be rolled back as above.

Because the round time periods for function send_my_let() and function rollback_nrt() are finite, the $MAX_LIVE$ time is finite, and the time between two neighboring works are finite, the above rollback operation will eventually take place.

**Property 3.4.** The LET, NRT, and UST tables will not grow indefinitely and any entry will be sent out or deleted eventually.

**Proof.** From Corollary 1, the speeds difference between LET filling and sending, and the limited down time, it is easy to see that the assertion for LET table is true.

The UST table is filled in by function manage_rpcs(). It is easy to see that the average length of the UST table is less than or equal to the average length of LET table because the UST contains only those RPC entries where the return value is US, while the LET contains the OK return entries. By Assumption 4, the assertion holds. From the assumption that the down time of any system entity is definite, we know that all RPCs issued by a host will be acknowledged and checked in function listen_extern_lets(). So, we can conclude that any entry in the table will be deleted eventually.

From Corollary 3 and its proof, it is easy to see that the assertion for NRT table is true.

4. PARALLEL RPC TRANSACTION MANAGEMENT

4.1 Algorithms

Parallel RPC transaction management can be provided by a small modification of the manager described in Section 3. In particular, only the function manage_rpcs() is modified. Otherwise, the manager has the same structure.

The modified function at first listens for a parallel RPC transaction call from the local host and processes them concurrently. If all of the RPCs return OK, or there are error returns, but no RPCs are
performed in the parallel call, then it simply tells the client the result and exits (to process new parallel calls). If there are some error returns and some RPCs are performed in the parallel call, then all performed RPCs (with OK returns) are put into the NRT table for rolling back, and all US return RPCs are put into the UST table waiting for clean up. It then tells the client that the transaction failed but that all performed RPCs are rolled back. The modified algorithm is provided by the following,

Algorithm 7.
manage_rpc_calls(ust_table)
UST ust_table[];
{
    UST *b;
    while (TRUE) {
        /* receive local RPC transaction calls */
        /* the process suspends until a transaction call comes */
        /* if more transaction calls come, they are queued up until */
        /* the processing of the current transaction call is over */
        receive(T = {P₁,...,Pₚ});
        /* concurrently execute the RPCs within the transaction */
        CBEGIN
        ret₁ = c(P₁),
        kᵢ = the destination host number,
        tᵢ = the time when issuing c(Pᵢ),
        i = 1,2,...,m;
        CEND;
        S_US = {Pᵢ|retᵢ = US};
        S_FL = {Pᵢ|retᵢ = FL};
        S_OK = {Pᵢ|retᵢ = OK};
        switch {
            case (S_US = S_FL = ∅):
                /* all RPCs executed and returned OK */
                /* tell the client the transaction returns OK */
                exit;
            case (S_OK = ∅ and S_US = ∅):
                /* error, but none RPCs are executed */
                /* tell the client the transaction returns FL */
                break;
            case (S_US! = ∅|S_FL! = ∅ && S_OK! = ∅):
                /* error, but some RPCs maybe executed */
                if (S_US! = ∅) {
                    for each Pᵢ ∈ S_US {
                        b.rpc = Pᵢ; b.host_to = kᵢ; b.time = tᵢ;
                        UST += b;
                    }
                }
                if (S_OK! = ∅) {
                    for each Pᵢ ∈ S_OK
                        NRT += Pᵢ;
                }
                /* tell the client the transaction returns US */
                break;
            /* while */
        }
    }
}

4.2 Properties
It is evident that Property 3.4 of Section 3.3 holds for parallel RPC transactions. The following properties can be easily established:

Property 4.1. If a parallel RPC transaction returns OK, all its RPCs have been executed and will not be rolled back.
Proof. The only place for the system to return OK
is when $S_{US} = S_{FL} = \emptyset$. In that case, all RPCs are executed correctly, and nothing is put into UST and NRT tables. So, they will not be rolled back.

**Property 4.2.** If a parallel RPC transaction returns $FL$, no RPC of the transaction has been executed.

*Proof.* The transaction returns $FL$ only if $S_{OK} = S_{US} = \emptyset$. That is, all the RPCs of the transaction return $FL$. By the definition of $FL$, no RPC of the transaction has been executed.

**Property 4.3.** If a parallel RPC transaction returns $US$, any executed RPCs will be rolled back.

*Proof.* The transaction returns $US$ only if $S_{US} \neq \emptyset$ or ($S_{FL} \neq \emptyset$ and $S_{OK} \neq \emptyset$). If $S_{US} \neq \emptyset$, Algorithm 7 then builds UST entries for all RPCs that return $US$ and inserts them into the UST. According to Property 3.4, those entries will be put into the NRT if their RPC have been executed. Again by Property 3.4, entries of the NRT will be rolled back eventually.

If ($S_{FL} \neq \emptyset$ and $S_{OK} \neq \emptyset$), Algorithm 7 then builds NRT entries for all RPCs that return $OK$ and inserts them into the NRT. Again by Property 3.4, they will be rolled back eventually.

5. IMPLEMENTATION ISSUES

We have implemented a preliminary version of the proposed RPC transaction manager. It currently runs on a network of SUN and HP workstations.

5.1 The SRPC System

Our experimental RPC transaction manager is built upon the Simple Remote Procedure Call System (SRPC) (Zhou, 1992). The SRPC is implemented using the TCP/IP protocol, and it only contains the essential features of an RPC system, such as a location server and a stub generator, among other things. The system is small, simple, expandable, and has facilities supporting rapid prototyping and fault-tolerant computing.

The SRPC has the following three components: A Location Server (LS), a system library, and a Stub and Driver Generator (SDG). This section describes the system architecture from a user's point of view.

Based on interface definition files, the SDG not only produces the server and client stubs, but also creates remote procedure framework, makefile, and driver programs for both server and client. After using the make utility, a user can test the program’s executability by simply executing the two driver programs.

From a programmer’s viewpoint, after the SDG compilation, the server part of an SRPC program consists of a server driver, a server stub, and a file which implements all the remote procedures (called procedure file). The server part (or a server program as it is sometimes called) is a “forever” running program which resides on a host and awaits calls from clients. The client part (or a client program) consists of a client driver and a client stub after the SDG compilation. It runs on a host (usually a different host from the server’s host) and makes calls to the server by using the remote procedures exported by the server.

When the client driver makes a call, it goes to the client stub. The client stub then, through the system library, makes use of the Internet entity of the client host to send the calling message to the Internet entity of the server’s host. At the server host side, the Internet entity will send the calling message to the server stub through the system library. The server stub then reports the call to the server and an appropriate procedure defined in the procedures file is executed. The result of the call follows the calling route in reverse, through the server stub, the system library and the Internet entity of the server host, the system library and the Internet entity of the client host, the client stub, and back to the client driver. This is called a direct call and the pre-condition of such a call is that the client knows the address of the server before the call.

With the help of the Location Server, the run-time address of a server can easily be accessed. One typical scenario of SRPC programs using LS can be described below: When the server is started, it first registers its location to the LS (through the server driver, server stub, the system library and the Internet entity of the server host, to the system library and the Internet entity of the LS host, the LS stub, finally to the LS driver) and then waits for clients to call. The clients know the server by a name (a character string) defined by the user in the interface definition files. When a client is invoked, it consults the LS for the server's location (through the client driver, client stub, the system library and the Internet entity of the client host, to the system library and the Internet entity of the LS host, the LS stub, finally to the LS driver). After the location is found, the client can then make any number of RPCs to that server by using the obtained location (as in a direct call). We name this calling as a typical calling. If a “shutdown” call is issued by a client program, the server un-registers itself from the LS and exits from the system.

Figure 7 depicts the system architecture using a typical RPC. The dashed line represents the RPCs from the user's viewpoint.
5.2 System Architecture

On each workstation, an RPC transaction manager server (RM) is invoked when the workstation is started. The RM uses Algorithm 1 to 7 described in Sections 3 and 4 (Algorithm 5 of Section 3 is replaced by Algorithm 7 of Section 4 in order for the RM to manage parallel RPC transactions). Currently, we use disk space as the stable storage. The three tables (LET, UST, and NRT) are then stored in disk files. This causes a longer time for logging the table entries (about 50 ms per entry on average).

If a client requires a service from a remote server, it needs to make the RPC to the server through two RMs as shown in Figure 8 (these calls are transparent to the user, of course). This arrangement requires one “remote” and two “local” RPCs as compared to one “remote” RPC in the original SRPC system (without the RMs). These redundancies are required to tolerate any network communication faults between the two hosts. Furthermore, the extra two “local” RPCs to the RMs will not consume much time as compared to the “remote” RPC.

The transaction manager must be able to perform the following recoveries in order to justify the redundancy added to the original SRPC system:

- **Transaction recovery.** This is the activity of ensuring atomicity in the presence of transaction aborts. The abort of a transaction can be due to system overloads, deadlocks, invalid inputs, and network communication faults.

- **Crash recovery.** The activity of ensuring atomicity in the presence of system crash.

The rollback operation is executed by the transaction manager to perform the above two recoveries.

5.3 System Tests

We have tested the system in various ways. An RPC monitor tool that we developed earlier (Zhou, 1993) was used to record the tracks of the RPCs. A “Send-and-Forward” application has been developed on the SRPC system to aid the development, testing, and implementation of the transaction manager (Lim and Zhou, 1993). In this application, the server acts as a message storage and clients act as both the message sender and receiver. There may be several such servers running on the network. The clients can then use our transaction calls to access multiple servers at the same time.

At first we tested the system when all the involved workstations and processes worked normally. The system worked correctly. Then we created some software errors for the client programs and server programs and let the RPCs execute incompletely. More specifically, we allowed the user to control the states (OK, FL and US) of the RPCs. These controls were made available at the server and the RPC transaction manager. The system actually rolled back...
the incomplete RPCs. The last test was to shut down a workstation manually and then re-start it after a while. The system also worked well in that situation.

From the above tests, we believe that the RPC transaction manager has the ability to manage the various RPCs and perform the necessary recoveries (transaction recovery and crash recovery) when faults arise.

6. REMARKS

The design and implementation of an RPC transaction manager is described in this paper. After a description of the problem and the RPC transaction models, we solve the RPC transaction management problem in two stages. At first, a system for managing transactions of single RPC calls is developed. Then, by extending the single RPC transaction management system, the parallel RPC transaction management system is described. In both cases, system properties are established (by informal arguments).

Almost all transaction management approaches use a two-phase protocol. It is easy to modify our model into a two-phase paradigm. In the first phase, the system will lock the affected data items (as we have done) and check if the involved RPCs will return OK or not. If the first stage is successful, the second phase will commit all the involved RPCs and release the locks. In that case, the system will be also able to deal with RPCs that are not capable of being rolled back in our sense. For example (refer to Section 2.1) the printing task will not be issued until the executability checking (checking if the RPC will return OK or not) is done. But the efficiency of this two-phase model will be less than our model, mainly because the executability checking will take considerable time, and the stable storage management in two-phase protocol is more complex than our model.

Our model is transparent to programmers. It acts as a run-time system within the programming environment. Programmers will not have much burden to maintain the RPC transactions in their programs. They can use RPC transaction calls almost as usual RPCs, and the system will do all the work. We feel this is better than the language level implementation.

Our model evolves from the "single RPC transaction management model" to the "parallel transaction management model." The recovery, rather than the concurrency, was the main focus during the evolution. Currently, the concurrency is supported by the UNIX heavy-weight processes through the fork system call and some parallel primitives (Zhou and Molinari, 1980). One of our next stages in our work is to use some light-weighted process package to provide more efficient concurrency.

Several extensions of the model are possible. For example, one may want to explore the nested RPC transaction model.

REFERENCES


386
A System for Managing Remote Procedure Call Transactions


Parallel Recovery in a Replicated Object Environment

Wanlei Zhou and Li Wang

School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
{wanlei, liwang}@deakin.edu.au

Abstract

A parallel recovery algorithm is proposed in this paper. It detects the parallelity between the recovery operations and executes them in parallel to achieve fast recovery for a network partition failure. An example is used to demonstrate the performance.

1 Introduction

Replication is the key to provide high availability, fault tolerance, and enhanced performance in a parallel and distributed processing system. However, a distributed processing system is subject to network partition failures (Cristian [2]). A network partition can separate a group of replica servers into subgroups in which the servers of the same group can communicate with each other, but not with the members of different groups. In some optimistic replication schemes, updates are allowed to perform within different partitions during a network partition failure period. This would lead to data inconsistencies between partitions which should be resolved when the partitions are repaired (Molesky [3] and Rabinovich [4]).

However, during a network recovery, accessing the affected replicas are suspended until the consistency of the replicas has been solved. Therefore a quick recovery is necessary, especially in a real-time system where the time of recovery is essential.

Suppose the network recovery is based on a log mechanism. One way in which this recovery can be made faster is to use semantic knowledge of operations to embed intelligence in the recovery algorithm. One of examples is the log transformation technique (Blaustein [1]) which seeks to classify operations into the following types:

- overwrite pairs
- commutative pairs
- conflicting pairs

The initial log is then simplified through successive log transformations that replace a pair of overwrite operations by the overwriting operation, pairs of operations that are inverse of each other and occur successively are canceled and so on.

The parallel merge protocol proposed by Tewari [5] exploits operations semantics further to gain significant savings over the log transformation technique. In this protocol, operations are classified into the following types:

- collapsible
  - associative
  - distributive
  - commutative
  - other non-arithmetic
- non-collapsible

The initial log is transformed into sequences of collapsible operations and then each sequence of collapsible operations is collapsed into one equivalent operation. For example, a sequence of write operations to the same object could be collapsed into one write operation to the object.

Unfortunately these existing recovery algorithms are sequential in nature. That is, they do not explore the parallelity of operations within a log.

Parallel processing is an effective way to achieve high response time and to meet real-time requirement. Parallel computing systems offer the promise of a quantum leap in the computing power. Therefore developing parallelly of a sequence of tasks will turn this promise into truth.

The main objective of this paper is to propose a recovery algorithm that can parallelly execute operations within a log, and therefore speed up the recovery process during a network partition repair.

The paper is organised as follows. Section 2 presents the model of object replicas, operations, and failure semantics through an example; Section 3 describes the algorithm for parallel recovery management; Section 4 discusses some further considerations to complete the proposed algorithm; Section 5 concludes the paper.
2 The parallel recovery model

We give the following definitions of the terms used in this paper to distinguish logs transformed at different stages.

- **A local log** is the sequence of operations performed at a replica site.

- **A global log** is the sequence of operations merged from all different partitions.

- **An initial merge log** is the sequence of operations that has to be executed at a replica site to bring up that replica into the global consistent state.

- **A final merge log** is the semantically transformed log for a replica site which will be actually executed to reach the global consistent state.

We also give the definition for what we mean about object replicas. We define our Object as a 4-tuple:

\[
OBJ = (ID_{obj}, DATASTRUCT_{obj}, OP_{obj}, P_{obj})
\]

where

- \(ID_{obj} = \{id_1, ..., id_n\}\), this means each object instance has its own identifier to distinguish it from others of the same class;

- \(DATASTRUCT_{obj} = \{e_1 : T_1, ..., e_n : T_n\}\), the constitutions the object has;

- \(OP_{obj} = \{OP_1, ..., OP_n\}\), the operations encapsulated with the object. An object is only accessible through these encapsulated operations;

- \(P_{obj}\) is a matrix used to represent the parallelity between the encapsulated operations. It will be detailed in the next section.

An object replica is simply an instance of the object with its own id.

Now we use an example to explain our parallel recovery mechanism in this section.

Assume we have an object queue. Without loss of generality, we assume that the queue stores integers only. There are four operations defined over the queue:

- \(AH(x: \text{ int in})\): add an integer \(x\) to the head of the queue;

- \(AT(x: \text{ int in})\): append an integer \(x\) to the tail of the queue;

- \(DH(x: \text{ int out})\): delete the head item and put its value into \(x\);

- \(DT(x: \text{ int out})\): delete the tail item and put its value into \(x\).

Therefore,

\[
DATASTRUCT_{queue} = \{q: \text{ seq of int}\};
\]

\[
OP_{queue} = \{AH, AT, DH, DT\}.
\]

Operations like \(AH(x)\) and \(DT(x)\) are not collapsible because there is no such an operation provided by the object to do both work. But we know that these two operations are mutually independent and can be executed in parallel. Similarly, if the \(AH(x)\) operation requires that only a single value \((x)\) be put into the queue through the operation, then it is impossible to collapse two successive operations (such as \(AH(x)\) and \(AH(y)\)) into one operation (such as \(AH(x+y)\)). In both cases, the parallel merge protocol proposed by Tewari [5] will not speed up the recovery process.

The parallelity of these operations can be defined as follows:

\[
AH(x) \parallel AT(x)
\]

\[
AH(x) \parallel DT(x) \mid \{ \text{ queue } \geq 1 \}
\]

\[
AT(x) \parallel DH(x) \mid \{ \text{ queue } \geq 1 \}
\]

\[
DT(x) \parallel DH(x) \mid \{ \text{ queue } > 1 \}
\]

where \(x \parallel y\) means that the two operations \(x\) and \(y\) can be executed in parallel, and \(x \parallel y\) \{condition\} means that \(x\) and \(y\) can be executed in parallel as long as the condition is true. \(|\text{ queue }|\) is the size of the queue.

Suppose that the network is partitioned into two parts, \(P_1\) and \(P_2\). Table 1 lists the operations on the queue while the network is partitioned. We assume that the queue contains two data items <0, 1> before the network is partitioned.

The first column of Table 1 shows the global sequence of all operations while the network is partitioned. The second column shows the operations in partition \(P_1\) and the third column shows the states of the queue in partition \(P_1\). The fourth column shows the operations in partition \(P_2\) and the fifth column shows the states of the queue in partition \(P_2\). The last column shows the states of the queue should the network stay un-partitioned.

During the network partition, the queue is updated by operations

\[S_1 = \{O_1, O_3, O_5, O_{10}, O_{11}, O_{15}, O_{16}\}\]

in partition \(P_1\). Similarly, the queue is also updated by operations

\[S_2 = \{O_2, O_4, O_6, O_7, O_8, O_9, O_{12}, O_{13}, O_{14}, O_{17}, O_{18}, O_{19}\}\]
<table>
<thead>
<tr>
<th>Step</th>
<th>P1</th>
<th>Queue in P1</th>
<th>P2</th>
<th>Queue in P2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>DT(x=104)</td>
<td>103 1 104</td>
<td>AH(202), AT(205)</td>
<td>202 1 201 205</td>
</tr>
<tr>
<td>1</td>
<td>AT(102), DH(x=103)</td>
<td>1 102</td>
<td>AT(206)</td>
<td>202 1 201 205 206</td>
</tr>
<tr>
<td>2</td>
<td>AH(0)</td>
<td>0 1 102</td>
<td>DT(x=205), AH(203)</td>
<td>203 202 1 201 205</td>
</tr>
<tr>
<td>3</td>
<td>AH(101), DT(x=102)</td>
<td>101 0 1</td>
<td>DT(x=205)</td>
<td>203 202 1 201 201</td>
</tr>
<tr>
<td>4</td>
<td>DH(x=101)</td>
<td>0 1</td>
<td>AT(204)</td>
<td>203 202 1 201 204</td>
</tr>
<tr>
<td>5</td>
<td>AH(101), AT(201)</td>
<td>101 0 1 201</td>
<td>DT(x=204), DH(x=203)</td>
<td>202 1 201 201 201</td>
</tr>
<tr>
<td>6</td>
<td>AT(102), DH(x=101)</td>
<td>0 1 201 102</td>
<td>DH(x=202)</td>
<td>1 201 201 201 201</td>
</tr>
<tr>
<td>7</td>
<td>AH(202)</td>
<td>202 0 1 201 102</td>
<td>AH(0), DT(x=201)</td>
<td>0 1 201 201 201</td>
</tr>
<tr>
<td>8</td>
<td>DH(x=202)</td>
<td>0 1 201 102</td>
<td>AH(101), AT(201)</td>
<td>101 0 1 201 201</td>
</tr>
<tr>
<td>9</td>
<td>AH(203), AT(204)</td>
<td>203 0 1 201 102 204</td>
<td>AT(102), DH(x=201)</td>
<td>0 1 201 201 201</td>
</tr>
<tr>
<td>10</td>
<td>DT(x=204), DH(x=203)</td>
<td>0 1 201 102</td>
<td>AH(202)</td>
<td>202 0 1 201 201 201</td>
</tr>
<tr>
<td>11</td>
<td>AH(103), AT(205)</td>
<td>103 0 1 201 102 205</td>
<td>DH(x=202)</td>
<td>0 1 201 201 201</td>
</tr>
<tr>
<td>12</td>
<td>DH(x=103), AT(206)</td>
<td>0 1 201 102 205 206</td>
<td>AH(203), AT(204)</td>
<td>203 0 1 201 201 204</td>
</tr>
<tr>
<td>13</td>
<td>DT(x=205)</td>
<td>0 1 201 102 205</td>
<td>DT(x=204), DH(x=203)</td>
<td>0 1 201 201 201</td>
</tr>
<tr>
<td>14</td>
<td>AT(104)</td>
<td>0 1 201 102 205 104</td>
<td>AH(103), AT(205)</td>
<td>103 0 1 201 102 205</td>
</tr>
<tr>
<td>15</td>
<td>DT(x=204)</td>
<td>0 1 201 102 205</td>
<td>DH(x=103), AT(206)</td>
<td>0 1 201 102 205 206</td>
</tr>
<tr>
<td>16</td>
<td>DT(x=205), DH(x=0)</td>
<td>1 201 102</td>
<td>DT(x=205)</td>
<td>0 1 201 102 205 206</td>
</tr>
<tr>
<td>17</td>
<td>DT(x=104)</td>
<td>0 1 201 102</td>
<td>AT(104)</td>
<td>0 1 201 102 205 104</td>
</tr>
<tr>
<td>18</td>
<td>DT(x=104)</td>
<td>0 1 201 102</td>
<td>DH(x=104)</td>
<td>0 1 201 102 205 206</td>
</tr>
<tr>
<td>19</td>
<td>DT(x=205), DH(x=0)</td>
<td>1 201 102</td>
<td>DT(x=205)</td>
<td>0 1 201 102 205 206</td>
</tr>
<tr>
<td>20</td>
<td>DT(x=205), DH(x=0)</td>
<td>1 201 102</td>
<td>DT(x=205)</td>
<td>0 1 201 102 205 206</td>
</tr>
</tbody>
</table>

Table 2: A possible parallel execution scheme

2. At each replica site, there is a mechanism to which parallel operations can be submitted. This mechanism is either implemented by the hardware or by a software simulation, such as in the cases of a parallel processing database system and a multi-processor system. This serves as the prerequisite of our protocol.

3. The object replica server needs to provide the inverse operation of each operation.

At the start of network partition repair, operations have been performed in different partitions are merged together according to the timestamps attached to each operation. The timestamps in different systems are converted into one standard time, for instance, the GMT (Greenwich Mean Time) universal time. This forms the global log, as for example, the sequence of operations O₁, ..., O₁₀ appeared in last section.

To form the initial merge log at each replica, it needs to add the inverse operation of each operation recorded in the local log in the reverse order and append the global log, L₁ and L₂ in last section are the instances of the initial merge log.

If causal ordering protocol is used in the replication system, this means that updates do not have to arrive at each replica in the same order. This turns out that initial merge logs might be different at each replica in the same partition. However, our proposed protocol is still applicable.

Now we can give the details about the matrix \( P_{obj}(i, j) \) mentioned in the last section. \( P_{obj}(i, j) \) is used to represent the parallelism of a set of operations provided by the object. Since \( P_{obj}(i, j) \) will have the same value of \( P_{obj}(j, i) \), only half of this matrix divided by the diagonal line is used to tell whether any two operations are executable in parallel. The other half is used to tell any two inverse operations are executable in parallel. \( P_{obj}(i, j) \) is set to value of 1 if \( Op_1 \) and \( Op_2 \) are parallel operations, otherwise the value is set to 0. Two operations are parallellable if they are mutually independent and not interacting with each other. The parallelism is derived from the implementation semantics of the object. Below is the matrix for the object queue introduced in last section.

\[
P_{queue} = \begin{bmatrix}
AH & AT & DH & DT \\
AH & 0 & 1 & 0 & 1 & AH \\
AT & 1 & 0 & 1 & 0 & AT \\
DH & 0 & 1 & 0 & 1 & DH \\
DT & 1 & 0 & 1 & 0 & DT \\
AH & 1 & 0 & 1 & 0 & DT \\
AT & 1 & 0 & 1 & 0 & DT \\
DH & 0 & 1 & 0 & 1 & DT \\
DT & 1 & 0 & 1 & 0 & DT \\
\end{bmatrix}
\]

For each entry in a log, an operation can be represented by a structure of \( (Op, Parallel, Tag, Timestamp, Next) \), Op is the operation name; Parallel has a value of 1 or 0 to mark the parallelism between a successive operations; Times-
<table>
<thead>
<tr>
<th>Sequence</th>
<th>P1</th>
<th>Queue in P1</th>
<th>P2</th>
<th>Queue in P2</th>
<th>Global Queue</th>
</tr>
</thead>
<tbody>
<tr>
<td>Initial</td>
<td>0 1</td>
<td></td>
<td>0 1</td>
<td></td>
<td>0 1</td>
</tr>
<tr>
<td>O₁</td>
<td>AH(101)</td>
<td>101 0 1</td>
<td></td>
<td></td>
<td>101 0 1</td>
</tr>
<tr>
<td>O₂</td>
<td>AT(201)</td>
<td></td>
<td>0 1 201</td>
<td></td>
<td>101 0 1 201</td>
</tr>
<tr>
<td>O₃</td>
<td>AT(102)</td>
<td>101 0 1 102</td>
<td></td>
<td></td>
<td>101 0 1 201 102</td>
</tr>
<tr>
<td>O₄</td>
<td>DH(x=0)</td>
<td></td>
<td>1 201</td>
<td></td>
<td>0 1 201 102</td>
</tr>
<tr>
<td>O₅</td>
<td>AH(202)</td>
<td></td>
<td>202 1 201</td>
<td></td>
<td>202 0 1 201 102</td>
</tr>
<tr>
<td>O₆</td>
<td>DH(x=101)</td>
<td>0 1 102</td>
<td></td>
<td></td>
<td>0 1 201 102</td>
</tr>
<tr>
<td>O₇</td>
<td>AH(203)</td>
<td></td>
<td>203 202 1 201</td>
<td></td>
<td>203 0 1 201 102</td>
</tr>
<tr>
<td>O₈</td>
<td>AT(204)</td>
<td></td>
<td>203 202 1 201 102</td>
<td></td>
<td>203 0 1 201 102 102</td>
</tr>
<tr>
<td>O₉</td>
<td>DT(x=204)</td>
<td></td>
<td>203 202 1 201</td>
<td></td>
<td>203 0 1 201 102</td>
</tr>
<tr>
<td>O₁₀</td>
<td>DH(x=0)</td>
<td>1 102</td>
<td></td>
<td></td>
<td>0 1 201 102</td>
</tr>
<tr>
<td>O₁₁</td>
<td>AH(103)</td>
<td>103 1 102</td>
<td></td>
<td></td>
<td>103 0 1 201 102</td>
</tr>
<tr>
<td>O₁₂</td>
<td>AT(205)</td>
<td></td>
<td>203 202 1 201 205</td>
<td></td>
<td>103 0 1 201 102 205</td>
</tr>
<tr>
<td>O₁₃</td>
<td>DH(x=203)</td>
<td></td>
<td>202 1 201 205</td>
<td></td>
<td>0 1 201 102 205</td>
</tr>
<tr>
<td>O₁₄</td>
<td>AT(206)</td>
<td></td>
<td>202 1 201 205 206</td>
<td></td>
<td>0 1 201 102 205 206</td>
</tr>
<tr>
<td>O₁₅</td>
<td>DT(x=102)</td>
<td>103 1</td>
<td></td>
<td></td>
<td>0 1 201 102</td>
</tr>
<tr>
<td>O₁₆</td>
<td>AT(104)</td>
<td>103 1 104</td>
<td></td>
<td></td>
<td>0 1 201 102 104</td>
</tr>
<tr>
<td>O₁₇</td>
<td>DT(x=206)</td>
<td></td>
<td>202 1 201 205</td>
<td></td>
<td>0 1 201 102 205</td>
</tr>
<tr>
<td>O₁₈</td>
<td>DT(x=205)</td>
<td></td>
<td>202 1 201</td>
<td></td>
<td>0 1 201 102</td>
</tr>
<tr>
<td>O₁₉</td>
<td>DH(x=202)</td>
<td></td>
<td>1 201</td>
<td></td>
<td>1 201 102</td>
</tr>
</tbody>
</table>

Table 1: Operations on the queue during the network partition

at the same period in partition P₂. When recovering from the network partition, the two sets of operations need to be brought together to produce a consistent queue.

The initial merge logs for P₁ and P₂ can be expressed as follows:

\[
L_{P₁} = \{O₁₆, O₁₅, O₁₁, O₁₀, O₆, O₃, O₁, O₂, \ldots, O₁₉\}
\]

and

\[
L_{P₂} = \{O₁₉, O₁₈, O₁₇, O₁₄, O₁₃, O₁₂, O₉, O₅, O₄, O₂, O₁, O₂, \ldots, O₁₉\}
\]

where \( \overline{O}_i \) represents the inverse operation of \( O_i \). It can be defined in terms of the original operations. For example, \( O₂ \) is \( AH(201) \), then \( \overline{O}_2 \) can be expressed as \( DH(x=201) \). Similarly, \( O₆ \) is \( DH(x=101) \), then \( \overline{O}_6 \) can be expressed as \( AH(101) \).

The two merge logs (\( L_{P₁} \) and \( L_{P₂} \)) need 26 and 31 steps to execute respectively. By applying the log transformation technique, we can reduce \( L_{P₁} \) to 24 steps (\( O₁ \) and \( O₁ \) can be canceled) and \( L_{P₂} \) to 29 steps (\( O₂ \) and \( O₁ \) are commutative and then \( \overline{O}_7 \) and \( O₂ \) can be canceled). This reduces the number of executing times for both logs to 92% and 94% of the executing times for initial merge logs, respectively. The parallel merge protocol cannot help in further reduction of executing times since all operations are non-collapsible.

However, by exploring the possible parallel operations, \( L_{P₁} \) can be reduced to 17 steps and \( L_{P₂} \) to 20 steps from the initial merge logs. Table 2 shows a possible parallel execution scheme for both merge logs.

The first column of Table 2 lists the sequence numbers of the merge operations. The second column shows the merge operations for partition P₁ and the third column lists the states of the queue in partition P₁. The fourth column shows the operations for partition P₂ and the fifth column lists the states of the queue in partition P₂. Operations in the same row (of the second and the fourth columns) are executed in parallel.

The above example illustrates that, by exploring the possible parallel execution within a merge log, we have reduced the times of executing both final merge logs to 65% of the initial merge logs.

3 The algorithm of parallel recovery management

This section gives a complete picture of how this parallel recovery mechanism works in general.

First we propose the following assumptions:

1. At each replica, there is a local log maintained by the server. The local log contains a list of operations performed by the server during a network partition period. We assume that the replicas are consistent at the beginning of the network partition, otherwise, we can start with the latest consistent state.
**tamp** is the time that the operation takes place locally; *Next* points to the next entry in the log.

Two procedures are designed to cut the initial merge log into subsequences, each subsequence of operations are executable in parallel.

1. **Init();**: initialize the matrix with the values of 0 or 1 for each element. If there is a condition under which two operations are parallelizable, this condition is embedded into the code, which is fulfilled by the routine *Detect.Parallel();* below.

2. **Detect.Parallel();**: cut the sequence of operations into subsequences which can be executed in parallel. Two successive operations are parallelizable if the corresponding value of the element in matrix *P* is 1 and these two operations satisfy the condition if there is any. A subsequence of more than two successive operations are parallelizable only if any two of the operations are mutually independent.

The C-like code for *Detect.Parallel();* as follows, *Init();* is omitted as initializing an array in C can be made at declaration spot:

```c
/* the matrix and the initialization */
bool P_obj[n-1][n-1] = { (...), ..., (...) };

/* the structure for the log entry */
struct Log_Entry {
  int Op;
  bool Parallel_Tag = false;
  int Timestamp;
  struct Log_Entry *Next;
}

Detect.Parallel(Init.Merge_Log)
struct Log_Entry *Init.Merge_Log;
{
  struct Log_Entry *SeqPtr, *SubSeqPtr;
  struct Log_Entry *SubSeqHead, *SubSeqTail;
  bool Tag = true;

  SeqPtr = Init.Merge_Log;
  while (SeqPtr) {
    SubSeqHead = SeqPtr;
    SubSeqHead->Parallel_Tag = Tag;
    SubSeqTail = SubSeqHead->Next;
    while (SubSeqPtr) {
      for (SubSeqPtr = SubSeqHead;
           SubSeqPtr != SubSeqTail;
           SubSeqPtr = SubSeqPtr->Next) {
        if (P_obj(SubSeqPtr->Op, SubSeqTail->Op) == 1) {
          if (SubSeqPtr == SubSeqTail) {
            SubSeqTail->Parallel_Tag = SubSeqHead->Parallel_Tag;
          }
          else {
            SeqPtr = SubSeqTail;
            Tag = not Tag;
            break;
          }
        }
      }
    }
  }
}
```

By submitting each initial merge log to the procedure *Detect.Parallel();*, the Parallel.tag of each entry will be set to value of 1 or 0. A successive sequence of operations with the same Parallel.tag value will be executable in parallel. As for an example, we apply *Detect.Parallel();* to the initial merge log *L*:

\[
L = \{ \overline{O_{16}}, \overline{O_{15}}, \overline{O_{11}}, \overline{O_{10}}, \overline{O_9}, \overline{O_{1}}, \overline{O_2}, \ldots, \overline{O_{19}} \}
\]

By looking at Table 1, *L* is equivalent to the sequence of *DT*, *AT*, *DH*, *AH*, *AH*, *DT*, *DH*, *AH*, *AT*, *DT*, *AT*, *DT*, *DT*, *DH*. After applying *Detect.Parallel();*, *L* will be divided into subsequences of the following which is the column *P* in Table 2:

<table>
<thead>
<tr>
<th>Parallel.tag</th>
<th>Parallelable Operations</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>DT</td>
</tr>
<tr>
<td>0</td>
<td>AT, DH</td>
</tr>
<tr>
<td>1</td>
<td>AH</td>
</tr>
<tr>
<td>0</td>
<td>AH, DT</td>
</tr>
<tr>
<td>1</td>
<td>DH</td>
</tr>
<tr>
<td>0</td>
<td>AH, AT</td>
</tr>
</tbody>
</table>

4 **Further Considerations**

Local recovery managers and the coordinator: At each object replica site, there is a local recovery manager (or server) which performs the job of bringing up the object replica to the global consistent state. There must be a coordinator who controls the whole process of the network recovery. One of local recovery managers running on a reliable machine can play the role of the coordinator. The coordinator receives the copies of local logs sent by the local managers, forms the global log, and sends it back to each site. At each replica, the local recovery manager forms the initial merge log by looking at its local log and the global log, then it invokes *ParallelDetect();*. Actually, *Init();* and *ParallelDetect();* are part of the local recovery manager.

The atomicity of the recovery process: The two-phase commit protocol is used by the coor-
5 Conclusion

Recently, Multiprocessor systems based on fast and inexpensive microprocessors are widely available. Making tasks parallelly executable will increase the throughput of the system. Our mechanism is at task level to achieve a coarse-grained parallelism.

This parallel recovery algorithm can be applied with the log transformation protocol (Blaustein [1]), the parallel merge protocol (Tewari [5]), and other semantics saving techniques for a log-based recovery of a network partition failure in a replicated object environment. The algorithm also can be applied on its own.

In this paper, we have presented an example queue to demonstrate the possible performance. The effectiveness of this algorithm majorly depends on the independency of operations provided by the object. As this is a very simple algorithm, the system cost of deciding the parallelizable subsequences could be omitted comparing with the operations actually performed.

Algorithm is complete and extensible to general situations by giving a few further considerations with respect to local recovery managers and the coordinator, the atomicity of the recovery process, and multiple objects at each site.

References


Distributed Object Replication in a Cluster of Workstations

Wanlei Zhou and Li Wang
School of Computing and Mathematics
Deakin University
Clayton, VIC 3168, Australia
Email: wanlei@deakin.edu.au

Abstract

This article is concerned mainly with the software aspect of building reliable and efficient services on a cluster of workstations. A key technology to achieve such a goal is service replication. However, designing and implementing replication system is a very difficult task. Based on an active replication model, this paper focuses on an object-oriented design pattern to simplify the design and implementation of distributed replications.

Key Words: distributed replication system, distributed object, cluster of workstations, fault-tolerance, group communication.

Introduction

Providing reliable and efficient services are the primary goals in designing a distributed system running on a cluster of workstations (COW). A distributed system manages many hardware/software components that are likely to fail eventually. In many cases, such failures may have disastrous results. With the ever increasing dependency being placed on distributed systems, the number of applications requiring fault tolerance is likely to increase.

Nowadays, we have seen a trend in distributed system design to shift from two-tier architectures or even monolithic architectures to three-tier or even n-tier architectures supported by the client/server model [3]. This means that a lot of services of a distributed system originally provided by a single piece of software are moved out of the kernel, forming individual servers. It is then crucial to guarantee that these servers will provide reliable and efficient services.

Service replication is a common means by which critical software and/or data are duplicated on a cluster of workstations. The goal of designing a replication system in general is to provide resilience and availability in the presence of failures while preserving one-copy consistency among all replicas. Service replication also provides a way of achieving better performance by distributing client requests to replicated servers to avoid the bottleneck in a single server environment.

Replication schemes generally follow two streams, namely primary-backup and active replication [8, 5]. Active replication is also referred as state machine approach [9]. There are various hybrids of above two replication models to suit application-specific requirements, such as the coordinator-cohort [2] and the semi-active replication [8]. This paper deals with the active replication schemes. We propose an active model where data consistency is defined by all operational replicas delivering the same set of update requests.

An active replication system uses the group communication concept. A lot of research has been done on group communication services. Isis [1] provides a completed toolkit based on process groups and the virtual synchrony concept for reliable distributed computing. Rachid [5] described how group services can support primary-backup and active replications. GARF [4] is a distributed object-oriented environment aimed at supporting the design and programming of reliable distributed applications through an extensive library of generic objects. The shortcoming of these systems is the lack of flexibility and the complexity of learning and use. The common object request broker architecture (CORBA) [7] is a platform generally for developing client-server distributed object computing model. However, fault tolerance was not a major concern in CORBA design.

This paper is aimed to promote distributed object
methodologies to bring flexibility and modularity into distributed replication systems. Based on the active replication model, we propose an object-oriented design pattern to simplify the design and implementation of distributed replications on COWs. Java is used in this paper as the target language for implementing the prototyped replication system. Through the experience of authors, it is indeed proven that Java is a neat and easy-to-work-with high-level language for developing programs running on COWs. Java-like pseudo code is used throughout this paper to demonstrate various design ideas.

The rest of the paper is organized as follows. Section 2 describes the object-oriented system model. Section 3 presents the group services to facilitate the design of the active replication scheme. Section 4 gives implementation overview. Section 5 summarizes the paper.

2 The Object-Oriented System Model

We model a distributed replication system as a group of replicas (or members, replica and member are used interchangeably in this paper) located on a cluster of workstations, only one replica at a workstation. The functionality of a replica is two-fold, providing a set of services and implementing the replication policy. The replication system as a whole should appear as a single service provider to clients.

![Figure 1. The active replication scheme](image)

Figure 1 depicts our active replication scheme. In this model, a client (c_i) is connected to only one member (R_j) of the group at a time. This member is responsible for processing and replying the client’s requests. An update request received at a member is multicasted to all other members before being delivered. A query request is delivered and processed locally. Data consistency is defined by all operational members delivering the same set of updates in FIFO order. Causal ordering and total ordering [1] of updates are not considered in the paper.

A replica is designed as an integration of two objects: the service implementation object (service object in Sh) and the replication management object, to separate the services provided and the replication policy adopted. The service object implements a set of services that are exposed to users. The service object is purely interested in proving correct services without being involved in any matter like where it should be located or whether it will be replicated. It is a distributed object that can be freely moved to any workstation. The granularity of service object could be as large as a database manager, as small as a query manager. Services provided are categorized as either query or updates. A query does not change the state of the service object but an update does.

The replication management object plays a role of being a surrogate to a service object. A set of replication management objects forms the object group that implements replication policies. Here we give Java-like skeleton codes for the service object (ServiceImpl), the replication management object (Surrogate), and the interface between them (ServiceInterface).

```java
public interface ServiceInterface { // implemented by any service object.
    public void upCall(Staring request);
}

class ServiceImpl implements Serviceinterface { // service provider.
    private variables;
    public service();
}

public ServiceA() {}
public ServiceB() {}
public ServiceC() {}
public ServiceD() {}

public String upCall(Staring services,Name,with-arguments) {
    switch (services.name) {
        case ServiceA: services();
        case ServiceB: services();
        case ServiceC: services();
        case ServiceD: services();
    }
}

class Deliver { // deliver client requests.
    public Deliver { // implementing the replication policy.
        private Deliver deliver();
        private Deliver deliver() { new Deliver();
    }
}

class Surrogate { // implementing service Interface.
    public Deliver(Surrogate service service) {
        public void deliver(Request request) {
            service.upCall(deliver, delivered);}
        public static void main() {
            Deliver service = new Deliver();
            Deliver deliver = new Deliver();
        }
    }

    public Deliver deliver() {
        Deliver deliver = new Deliver();
    }
}
```

Through the interface upCall(), any service object implementing the upCall() can be integrated with the underlying replication system (surrogate object) without any internal modifications involved in both objects. By adopting this interface, different service objects are able to plug-and-play. If a service object needs to be upgraded, the surrogate is not affected, and vice versa.
The replica group is designed to be dynamic in the sense they do not operate on a static set of workstations, rather, any workstation can be used. The group membership of the replication system is changed under three kinds of events: join, leave, and fail. A join event is issued by a new member joining the group, or by a member re-joining. A leave event happens when a member leaving voluntarily. A fail event is due to a member’s crash failure. The next section will discuss what property has to be preserved during the membership changes.

A name server (NS) is established to sit between clients and the group. We assume the NS is installed at a stable workstation. By doing so, the replication group is isolated from clients; clients do not need to keep the membership of the group, rather only have to know the address of the NS and the symbolic name of the replication group. Upon a request to the NS for a connection to a replicated server, a client is bound with a member of the group by the NS. Then the client makes the direct connection to that member which becomes the accessing point of the replicated service. When that member becomes unavailable, the client asks the NS for the next available member. Membership changes are informed by the replication system to the NS. The dynamic membership of the group is passed through the NS to clients in a dynamic and transparent way.

The NS can employ some load balancing policy to assign group members to clients so that loads of group members are roughly even. We form a ring of operational members in the name server, a client is allocated with the next member in the ring. Thus the clients are evenly allocated to members of the group.

Here we give the NS interface. This interface can be used to manage multiple replication groups.

```java
public interface NameService {
  void create(String symbolic name, Setting groups); //create group
  void delete(String symbolic name) throws GroupNotFoundException;
  void addMember(String symbolic name, GroupMember) throws GroupMemberNotJoinedException;
  void delMember(String symbolic name, GroupMember) throws GroupMemberNotJoinedException;
  void disConnected(String symbolic name, GroupMember) throws GroupMemberNotJoinedException;
  void isMember(String symbolic name, GroupMember) throws GroupMemberNotJoinedException;
  void listMembers(String symbolic name) throws GroupMemberNotJoinedException;
  GroupMember findMember(String symbolic name, String memberID) throws GroupMemberNotJoinedException;
  boolean isEmpty() throws GroupMemberNotJoinedException;
}
```

3 Group Communication Services and Replication Consistency

Without considering failures, the implementation of group services is trivial. However, group services need to be fault-tolerant to provide expected services in the face of failures.

Any group member may fail then becomes unavailable. A failure can be caused by a number of reasons. (1) Voluntary-exit. A member exits voluntarily due to, such as lacking of resources, being switched off by the system administrator, etc. In this case it can restart at a well-defined state. (2) Process crash. A member process stops the execution due to various reasons, such as being killed deliberately. It may not be able to restart at a well-defined state. (3) Site crash. The workstation a member is running on has crashed. The member definitely can not restart at a well-defined state. We do not consider network partitioning failures [6] here. This problem is dealt with in another paper by one of the authors [10].

The basic idea of the multicasting atomicity protocol is to let operational members exchange (re-multicast) their last received message from the failed member. Let \( G = \{g_1, \ldots, g_n\} \) be the members of the group. At \( g_i \), it keeps a vector of the latest received multicast from every other member, \( last\_mcast[g_i] = m_{i}^{k}, \) where \( k = 1, \ldots, n \). In the absence of failures, upon reception of \( m_{k}^{k} \), \( g_i \) updates its vector, i.e. \( last\_mcast[g_i] := m_{i}^{k} \).

When a member, say \( g_i \), first detects the failure of \( g_k \), it initiates the multicasting atomicity protocol by multicasting its \( last\_mcast[g_i] \), that is, \( g_i \) multicasts a \( m_{atom}(g_k, m_{i}^{k}) \) indicating \( g_k \) has failed and attaching the latest received \( m_{i}^{k} \) from \( g_k \). Upon receiving \( m_{atom}(g_k, m_{i}^{k}) \), a member updates its \( last\_mcast \) vector according to the following algorithm:

**Multicasting atomicity protocol**

At each operational member \( g_i \):

- On receiving \( m_{i}^{k} \):
  
  \[
  last\_mcast[g_i] := m_{i}^{k}.
  \]

- On detecting the failure of \( g_k \):
  
  Multicast \( m_{atom}(g_k, last\_mcast[g_k]) \)

On receiving \( m_{atom}(g_k, m_{i}^{k}) \):

```
if last\_mcast[g_k] = m_{i}^{k-1}:
    then update last\_mcast[g_k] := m_{i}^{k};
else if the case of last\_mcast[g_k] = m_{i}^{k} or
    last\_mcast[g_k] = m_{i}^{0};
    ignore m_{atom}(g_k, m_{i}^{k});
end if
if g_k has not sent a m_{atom}
    then Multicast m_{atom}(g_k, last\_mcast[g_k])
end if
```
During the execution of the multicasting atomicity protocol, it is likely there might be other failures. For brevity, let $g_1$ be the first one that fails, then G becomes $\{g_2, \ldots, g_n\}$, and $m^{21}_i$ is the message unstable in $\{g_2, \ldots, g_n\}$. Multicasting atomicity protocol is performed by members in $\{g_2, \ldots, g_n\}$ to propagate $m^{21}_i$. During the execution, say, $g_2$ holds the $m^{21}_i$, then multicasts $m^{atom}_i = (g_1, m^{21}_i)$. If $g_2$ fails during the multicasting, it leaves $G = \{g_3, \ldots, g_n\}$. Thus $m^{21}_i$ has to be recovered in $\{g_3, \ldots, g_n\}$. This procedure may go on if there are further failures until $G = \{g_n\}$. No matter whether $g_n$ has got $m^{21}_i$ or not, the atomicity is satisfied.

Membership is the information about a group that shows who is in the group. The membership is changed by a member's join/leave/fail event. In essence, the membership is replicated at each member and also the NS, so a multicast can be sent to those who are currently operational, and also clients can connect to new members through the NS.

The membership is represented by a view consisting of members of $\{g_1, \ldots, g_n\}$ and is encapsulated by operations, leaveGroup() and joinGroup(). Members start with the same local view, view0. Let view0 = 0, 1, ..., be the successive views of the group, viewi, and viewi+1 are different only by an addition or a deletion of one member. A view update message $m_{join}$, $m_{leave}$, or $m_{fail}$ is multicasted to members in viewi ∩ viewi+1, changing viewi to viewi+1.

View changes are delivered in total-order at all members. Total ordering is accomplished by designing a member as the coordinator. A join/leave/fail request is sent to the coordinator from where it is multicasted to all other members. If the coordinator fails, the new coordinator is selected by an algorithm, for instance, the oldest member in the group.

Member joining. A joining event is a concurrent event with multicasting events among group members, a scenario depicted by Figure 2. Where $g_4$ is the new member joining the group. At the same time, a multicast $m_1$ by $g_1$ is sent to current view $\{g_1, g_2, g_3\}$, which is delivered at $g_2$ and $g_3$ before $m_{join}$ but after $m_{join}$ at $g_1$. Suppose $g_1$ is the coordinator, it transfers its state to the new member at the instant of receiving $m_{join}$. Later on, when $g_1$ delivers $m_1$, $g_4$ will miss it.

Virtual synchrony on membership service. The virtual synchrony model [1] developed by Isis project defines the membership atomicity, which is a property that messages sent in viewi have to be delivered in the same viewi. That is, a $m_{join}$, $m_{leave}$, or $m_{fail}$ is delivered at all members viewi when messages sent in viewi are group-wide safe. Membership atomicity is implemented by the flush proc [1].

Member leaving. Conceptually, the flush protocol is needed when a leaving event ($m_{leave}$ or $m_{fail}$) happens, state transfer is not involved. But flush protocol can be as a checkpointing mechanism that a member leaving unearthy can have a consistent state with the group at the point of departure. Later, this member can rejoin the group by catching up on missed operations. The catch-up might be a more appropriate solution than state-transfer in the situation where the group manages a large volume of data.

When dealing with a member leaving by crash, both multicasting atomicity protocol and flush protocol have to Multicasting atomicity protocol stabilizes the last multi-view of the failed member which should be delivered to all members before the membership is updated. Therefore, if two protocols can be combined in one go by letting $m_{fail}$ carry the unstable multicast of the failed member.

The membership consistency between the group and NS. A member designated as the coordinator is responsible for updating the NS right after its local view is updated. Hence, the NS executes the same set of membership updates in the same order as the group.

4 System Implementation

There are three components in the system, replica clients, and the name server. Figure 3 depicts the relationship of these components. A replica is composed of two objects, the service object and the surrogate object. We have presented the skeleton codes for the service object and at the name service interface in Section 2. The most complex object is the surrogate object. This section is focused on the internal design of the surrogate object, which is a coherent integration of many other objects.
A member is fully connected with all other members. The replicated server group can be created in a number of ways: (1) The group is created by taking a list of members as the input. (2) The group is created with empty member, then each member joins the group.

The surrogate has the following state variables:

\[ \text{member.state} = \{ G, \text{InQ}, \text{OutQ}, \text{last.mcast}[G] \} \]

\( G \) represents the local view of the group membership. \( \text{InQ} \) stores all incoming messages from both clients and other members of the group. Requests are delivered in FIFO ordering. \( \text{OutQ} \) stores outgoing replies. \( \text{last.mcast}[G] \) is used by the multicasting atomicity protocol.

Each surrogate runs multiple threads to handle different matters. There are three types of threads, client-connection thread, multicast-reading thread, and the main thread. A member starts a thread for accepting clients' connections. This thread spawns a child thread for each client connection. A child thread is used to read the client requests, deposit them to \( \text{InQ} \), get replies from \( \text{OutQ} \), and write them back to the client. A multicast-reading thread is setup to read all propagated messages from peer members of the group. The main thread controls the whole process of replication management by delivering requests from \( \text{InQ} \), issuing up-calls to the service object, and putting them to \( \text{OutQ} \).

Messages are formed by using a three-tuple format:

\[ \text{Message} ::= (\text{type}, \text{mid}, \text{message.content}) \]

\( \text{type} ::= \{ \text{u}, \text{q}, \text{p}, \text{j}, \text{t}, \text{n}, \text{r} \} \)

\( \text{mid} ::= \text{sender-network-address.message-sequence-no} \)

\( \text{message.content} ::= \text{the.content.of.the.request} \)

Messages are typed in order to direct different actions.

A surrogate is an integration of many auxiliary objects. Here we describe in detail how the surrogate object is constructed.

- **class Group.** A group object manages the local view of the operational members. Basically, it provides two methods, \( \text{leaveGroup()} \), and \( \text{joinGroup()} \), to remove a member and to add a member respectively.

- **class Message.** This object forms messages used in the system by keeping a local counter for attaching a sequence number to each multicasted message.

- **class Connnmembers.** This object sets up the stream-based connection to each member in the group. Streams are collected into a vector, \( \text{channels} \), which is used for multicasting.

- **class Deliver.** This object delivers requests being added to \( \text{InQ} \). In the normal execution (not performing any protocol), messages are delivered in FIFO ordering.

- **class Flush and class McastAtomAndFlush.** A \( \text{Flush} \) object is used to perform the flush protocol. A \( \text{McastAtomAndFlush} \) object performs the multicasting atomicity protocol and the flush protocol.

- **thread classes.** An \( \text{AcceptClients} \) object is a thread for accepting client connections. When there is a connection from a client, it spawns a child thread to that client which is a thread of \( \text{AcceptClient} \). A \( \text{McastReading} \) is a thread of reading all multicasts from peer members so that the main thread can be free from this matter to concentrate on request handling.

- **class Surrogate.** An instance of surrogate is a replica which runs on a workstation. A surrogate object is the control object of all the other objects.
5 Remarks

A design pattern for developing distributed object replications in COWs has been presented. The pattern separates a replicated and distributed object into two interrelated components, the service implementation object (the service object) and the replication management object (the surrogate). An interface between these two objects is through the public interface ServiceInterface containing a method upCall(). The surrogate object manages the connections with other surrogates and clients; it intercepts clients' requests, propagates update requests to other surrogates, and issues up-calls to the service object; it executes protocols in the event of a member's join/leave/fail. To facilitate the design of the active replication model, two group communication services, multicasting atomicity and membership atomicity, are embedded in the surrogate object. The atomic multicasting service guarantees members receiving the same history of updates. Whereas the dynamic membership service guarantees that membership changes do not bring inconsistency to the replication system. The pattern is general enough to the design of any object replication system.

References


Automating the Construction of Replicated Objects in a Cluster of Workstations

Wanlei Zhou
School of Computing and Mathematics, Deakin University, Clayton, VIC 3168, Australia,
Email: wanlei@deakin.edu.au

Li Wang
Phoneware Communication Systems, Burwood East, VIC 3151, Australia

Abstract

This paper describes a tool that mimics the design of the remote procedure call (RPC) system to support the building of replicated objects in a cluster of workstations (COW). The tool includes an interface definition language for describing a replica group, a language preprocessor and a runtime library system. The paper also presents an example and discusses some performance issues for replicated objects.

1 Introduction

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a cluster of workstations. However, building such a system remains as a difficult and challenging task, mainly due to the difficulty of maintaining data consistency among replicas and the lack of easy and efficient tools supporting the development procedure.

Traditionally, the remote procedure call system helps building the client/server systems very well when the server is a single entity [2]. However, when it comes to build a replicated server system, the RPC system does not provide sufficient support. The reason is that the RPC system only provides point-to-point communications between the client and the server. Whereas in the replicated server system, the client has to communicate with a group of replicas, and replicas have to communicate with other replicas intensively upon each update request sent out from a client to guarantee the data consistency among replicas.

There are also many research projects working on the scope of deploying object-oriented methodologies in building reliable distributed systems. They are: Arjuna [7], Espirit Delta4 [6], Horus [8], GARF [1], Electra [5], FRIENDS [4], etc. However, most of these toolkits only provide library-level functions and protocols, therefore programming based on these toolkits is still manual and programmers have to have intimate knowledge about the functions and protocols provided by the toolkits.

The purpose of this paper is then to develop a toolkit aimed at automating the developing procedure of replication systems. The toolkit includes an interface definition language for defining a replica group, a preprocessor that generates a set of modules for constructing client and replica programs, and a runtime system that supports the executions of client and replica programs. The implementation is based on Java and fully object-oriented designs are enforced throughout the toolkit development.

2 The Object-Based Execution Model

Our replication scheme is named as the primary-peer replication scheme (PPRS) [10], as depicted in Figure 1. The scheme is based on active replication [3], but one replica is designated as the primary, others as peers. A primary-backup group is configured simply by directing all requests to the primary of the group and letting peer members (now backups) only receive propagations.

![Figure 1. The Primary-Peer Replication Scheme](image)

From Figure 1, we can extract two basic system components, namely client (C1) and replica (R1). To separate concerns, we use an execution model that mimics the RPC systems, where a client is supported by a client stub and a server is supported by a server stub. In our executional system, a client is supported by a client proxy and a replica (replicated server) is supported by a replica surrogate. In essence,
an executable replication system consists of three types of software entities: client, replica, and the group management server (GMS). Figure 2 depicts the relationship among these entities. These software entities are represented and implemented by objects.

![Figure 2. The Execution Model](image)

### 3 The Constructing Tool

In essence, the tool for constructing a replicated service system based on the active replication scheme and the object-based execution model includes three components: the replica group interface definition language (RGIDL), the RGIDL preprocessor and a runtime system.

#### 3.1 The RGIDL Language

The RGIDL language is used to specify a replica group. The declaration about a replica group includes group name, group style, initial members (if there is any), update ordering constraints, and a service interface embracing a set of operations exported to clients. Table 1 shows the RGIDL syntax in EBNF:

<table>
<thead>
<tr>
<th>Rule</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>RGIDL</td>
<td><code>GroupDef { GroupName (GroupName) }</code></td>
</tr>
<tr>
<td>GroupDef</td>
<td><code>GroupName = GroupName</code></td>
</tr>
<tr>
<td>GroupList</td>
<td>`GroupList = GroupName (GroupName)</td>
</tr>
<tr>
<td>MemberList</td>
<td>`MemberList = MemberName (MemberName)</td>
</tr>
<tr>
<td>MemberName</td>
<td><code>MemberName = MemberName</code></td>
</tr>
<tr>
<td>MemberAttr</td>
<td><code>MemberAttr = AttrName (attrValue)</code></td>
</tr>
<tr>
<td>AttrName</td>
<td><code>AttrName = AttrName</code></td>
</tr>
<tr>
<td>attrValue</td>
<td><code>attrValue = attrValue</code></td>
</tr>
<tr>
<td>Group</td>
<td><code>Group = GroupName (GroupName)</code></td>
</tr>
<tr>
<td>Preprocessor</td>
<td><code>Preprocessor = Preprocessor</code></td>
</tr>
<tr>
<td>Preprocessor</td>
<td><code>Preprocessor = Preprocessor</code></td>
</tr>
<tr>
<td>Dynamic</td>
<td><code>Dynamic = Dynamic</code></td>
</tr>
<tr>
<td>DynamicAttr</td>
<td><code>DynamicAttr = AttrName (attrValue)</code></td>
</tr>
<tr>
<td>attrValue</td>
<td><code>attrValue = attrValue</code></td>
</tr>
<tr>
<td>Interface</td>
<td><code>Interface = Interface</code></td>
</tr>
<tr>
<td>InterfaceAttr</td>
<td><code>InterfaceAttr = AttrName (attrValue)</code></td>
</tr>
<tr>
<td>attrValue</td>
<td><code>attrValue = attrValue</code></td>
</tr>
</tbody>
</table>

**Table 1. The RGIDL syntax in EBNF**

A replication system is defined by two parts, a group definition part and a service interface definition part.

- **The group part.** It describes a replica group, such as the **GroupName**, the **Group style** (PP and PB), the **MemberList**, the host name and port number of the **NS**, the propagation frequency, etc.

- **The service interface part.** It defines a set of operations that the server exports to clients.

#### 3.2 The Preprocessor and Runtime System

With a given RGIDL specification file, the preprocessor generates three sets of files: (1) the service interface file containing a set of Java methods (operations exported to clients); (2) client driver (client object) and client proxy files to form the client program; and (3) server driver (service implementation object) and server surrogate files to form the replica program. Figure 3 depicts this procedure.

![Figure 3. The preprocessor](image)

The runtime system supports the execution of both client programs and replica programs. The runtime system helps achieving server distribution and replication transparency. The design principle for the runtime system is to layer objects so that high layer objects are based on and supported by the lower layer objects. The runtime system is developed on the basis of Java networking package and itself is a package consisting of a collection of objects: utility objects, mini-protocol objects, root proxy and root surrogate. Figure 4 depicts the layers of the runtime object model.

![Figure 4. The service object model](image)

Utility objects manage basic resources, such as the replica group membership, message formats being transmitted among system entities (clients and replicas) to direct different actions, queues for buffering incoming and outgoing messages, communication channels established between clients and replicas, sites and port numbers on which replicas will be running, etc.

Protocols used to handle data consistency are implemented as mini-protocol objects. This layer is built above the utility object layer. It contains three mini-protocols.
at the moment: CrashAtomicity, StateTransfer and VoluntaryLeave. CrashAtomicity handles a replica crashing event, StateTransfer handles a new replica joining event, and VoluntaryLeave handles an event of a replica leaving voluntarily.

The proxy object is an extended client side stub that handles communications with the NS and the replicated server group. The proxy object in the runtime library is the root proxy object and is supposed to be inherited by each individual client invoking a specific service.

The surrogate object is an extended server side stub which not only handles the communications with clients but also keeps the NS updated with membership changes, and communicates with other surrogates to implement the replication control protocol. The surrogate object in the runtime library is the root surrogate object to be inherited by all particular replica surrogates. The root surrogate is designed by employing the following ideas: (1). Communications are based on reliable TCP/IP channels. (2). Multi-threading is used to improve the performance of the surrogate. Clients can make connections at any time, each client is handled by a separate thread which is spawned at the connection time to receive and reply the client requests. Communications among replicas of the group are handled by a separate thread and a few other threads are established to handle concurrent matters in the surrogate. (3). The surrogate object is a composite/aggregation of mini-protocol objects and utility objects.

4 An example

Suppose we need a simple host name service (HNS) that stores mappings between symbolic host names and their IP addresses of a local domain. To be able to provide continuing HNS in the presence of a crash failure, we replicate the HNS to a primary-backup group. Without loss of generality, we assume one backup is employed. Assume the following operations are provided by the HNS server:

1. addHost (String HostName, String IPAddress). An update operation which adds a new host to the local domain.
2. updateHostName (String OldHost, String NewHostName). An update operation that changes an old hostname to a new one.
3. deleteHost (String HostName). An update operation that removes a host from the local domain.
4. hostName (String IPAddress). A query that returns the host name that has the IP address.
5. IPAddress (String HostName). A query that returns the IP address of the host.

The RGRIDL file HNS.idl defines the group and the service interface:

The specification declares a primary-backup group named HNSGroup. The primary will be running on machine bofur.cm.deakin.edu.au, and the backup will be running on gallum.cm.deakin.edu.au, both are running at the same port number of 1040. The GMS server is running on alice.cm.deakin.edu.au at port 1050. The propagation frequency is every 1 request from the primary to the backup. Figure 5 shows the procedure of using the preprocessor. The preprocessor generates a set of files according to the RGRIDL definition file, HNS.idl:

- HNS.java — the service interface file. It defines the set of operations.
  ```java
  public interface HNS {
    public void addHost (String HostName, String IPAddress);
    public void updateHostName (String OldHost, String NewHostName);
    public void deleteHost (String HostName);
    public String hostName (String IPAddress);
    public String IPAddress (String HostName);
  }
  ```

- HNSProxy.java — the client proxy file. The file contains the HNS proxy object which is an extension to the root proxy in the runtime system and implements the HNS service interface:
  ```java
  public final class HNSProxy extends PPRS.Proxy implements HNS {...}
  ```

The HNSProxy first queries the GMS to get the reference to a primary, then it makes the direct connection to the primary. When the primary is down, HNSProxy queries the GMS to get the backup's reference and switches to the backup.

- HNScli.java — the client driver file. The file contains the client object that instantiates HNSProxy and then invokes any operation provided by the HNS server.

- HNSSrv.java — the server surrogate file. The file contains the HNS surrogate which is an extension to the root surrogate of the runtime system.
public class HNSSurrogate extends PFSSPB surrogate {...

- HNSSer.java — the server driver file. The file contains the implementation of the HNS interface and the server driver that instantiates and starts up the HNSSurrogate.

Two executable files HNScli.class and HNSSer.class are generated after the compilation. Both the primary and the backup use the same code fragments of service object and server surrogate but start the execution in different roles: java HNSSer 0 denoting the primary and java HNSSer 1 denoting the backup. When the primary and the backup have established the connection to each other and are ready to accept requests, clients can be started to send requests.

usage: HNSSer 0/1 select the primary/backup
usage: HNScli/1 select the client

5 Performance Evaluation

5.1 Metrics

The evaluation is based on two metrics: the average response time (ART) over requests, and the average system throughput (AST) which is the average number of requests that can be executed by the system.

Assume we have an n-replica system: R^n, where R^n = \{R_1, ..., R_n\}. To evaluate the overall ART of the replication system, assuming the ART at each replica is ART(R_i), where 1 \leq i \leq n, then we define the overall ART to be the average value of average response times achieved at each replica:

\[
ART_{R^n} = \frac{1}{n} \sum_{i=1}^{n} ART(R_i)
\]  

To evaluate the overall AST, we assume that N requests are received by each replica of the group. The replicated system starts from a global consistent state. By executing all n*N at all replicas, the replicated service system stops at another globally consistent state. Let AST(R_i) represent the AST at R_i; it is measured by the average number of requests executed at R_i to finish all n*N requests, then we can define the overall AST to the minimum value among all ASTs achieved by each replica:

\[
AST_{R^n} = \min(\text{AST}_{R_1}, ..., \text{AST}_{R_n})
\]

That is, the overall system throughput is determined by the replica who is the last one to finish executing all n*N requests. However, these two metrics can be largely affected by many factors, including the following:

- Replication degree. This relates to the number of replicas employed in the system. The response time over queries can be greatly improved if more replicas are involved. However, the response time over update operations is complicated. The response time also depends on other factors, such as the update percentage rate and the strength level of ordering constraints being placed on the set of update operations. One thing we can be sure is that, the more replicas involved, the more network communication traffic is generated.

- The number of clients connected to the replicated system. In general, the more clients connected to the system at the same time, the slower the response time is. However, if each client only sends requests in a very slow pace, i.e., the request arrival rate is very low, then the response time should not be significantly affected.

- Update percentage. Service replication systems are very useful and efficient in an application environment where the percentage of update operations is low. If the percentage of updates is high, the performance of the replicated system can be worse than that of the non-replicated system. Thus, the higher update percentage rate is, the less efficient (both the response time and system throughput) the replicated system becomes.

- Propagation frequency. If the propagation happens more often, then the size of the propagation message is tiny. The response time is affected by relatively tiny delays at short time intervals, and the delay is the result of sending out the propagation message by the underlying communication primitive. When the propagation happens less frequently, the propagation message grows large. The response time is affected by relatively large delays but at longer time intervals.

Thus, the ART is determined by whoever becomes the prominent factor—the more frequent tiny delays or less frequent large delays.

In this paper, we concentrate on the study of two metrics, the ART and the AST, with respect to propagation frequencies, but fixing the replication degree to four replicas and update percentage to 100%.

Instead of calculating the ART and AST, we calculate \{ART_{R_1}, ..., ART_{R_n}\} and \{AST_{R_1}, ..., AST_{R_n}\} to show the detailed testing results on different machines.

5.2 The System Setting

In our experiments, we allocated four Sun Sparc stations for running four replicas. These four Sun stations have slightly different hardware configurations. Each replica is fully connected with other three replicas by TCP/IP reliable streams. When a replica is first started, according to its
5.3 The Effect of Propagation Frequency

This experiment is to study how the propagation frequency affects the response time and the system performance. Propagation frequency affects a system where most operations are commutative or causal. This is because if an operation is a total or total-causal operation, the propagation is triggered right away. In other words, total or total-causal operations break up the regularity of propagations at the specified frequency.

In this particular experiment, we assume that all eight operations are commutative operations so that the propagations happen exactly at the frequency specified. We set the propagation frequency vary at: (1) every 1 request; (2) every 2 requests; (3) every 3 requests; (4) every 10 requests; (5) every 20 requests; (6) every 50 requests.

The test also employed asynchronous propagation method, which means a request is handled and replied to the client right away. In contrast, the synchronous prope...
gation requires a propagation to reach peer replicas before the request is handled and returned to client. Asynchronous propagation normally gives a quicker response time.

Four Sun stations involved in this test are: “bofus”, “bifus”, “durin” and “elwing”. Since each of them has a slightly different hardware configuration from others, in turn each machine has shown a slightly different performance.

Figure 6 shows the testing results of ARTs over 100 requests at varying propagation frequency rates on four machines. From the figure, we can observe that at the frequency of every 5 and 10 requests, the ARTs are at the lowest level (the best). When propagation happens more frequently, i.e. less than every 5 requests, the ARTs tend to be higher. When the propagation happens less frequently, i.e. every more than 10 requests, the ARTs tend to grow slightly. This can be explained by the fact that the ART is determined by whoever becomes the prominent factor, either the more frequent tiny delays or less frequent large delays, as we discussed in the introduction section under propagation frequency.

Figure 7 shows what the ASTs are achieved on four machines. At the frequency of every 5 and 10 requests, replicas have relatively high throughput rates. This matches the test results depicted by Figure 6, where the ARTs at the frequency of 5 or 10 are the lowest, in turn, highest throughput rates should be achieved there.

In the implementation, delivering client requests and delivering received propagations are handled by two separate child threads in parallel as well. Threads can be set to have a high/low priority. The experiment shown in Figure 6 is organised by setting the thread of delivering client requests to have a higher priority than that of delivering received propagations.

We also conducted an experiment by giving all threads an equal priority, the result shown in Figure 8. The ARTs are affected by this setting quite significantly. Also we can observe that the lowest ARTs are shifted to frequencies at 10 and 20. Figure 9 shows the comparison in ARTs of these two experiments on the same machine “durin”. The top curve represents the test result of assigning all threads an equal priority, whereas the bottom curve represents the test result of assigning the delivering-client-request thread a higher priority. The results demonstrate, by giving client requests a high priority, the ART is shortened quite dramatically.

6 Remarks

This paper presents a toolkit for automating the development of distributed replication systems on COWs. An example and a performance study are also described in the paper. The toolkit uses a proxy object to support a client and a surrogate object to support a replica. The RGIIDL language is proposed for defining a replicated server group and a set of remote operations. A preprocessing generator of the relative proxy and surrogate objects to form the client and the replica side programs. The constructing tool allows the developer of a replicated service system to concentrate on the implementation of service programs and client interface programs while leaving the replication control protocol to be generated by the tool. This can shorten the development life-cycle and reduce the complexity significantly.

References


Update Ordering in Distributed Replication Systems

Wanlei Zhou
School of Computing and Mathematics, Deakin University, Clayton, VIC 3168, Australia, Email: wanlei@deakin.edu.au

Li Wang
Phoneware Communication Systems, Burwood East, VIC 3151, Australia

Abstract

We propose a model for update orderings and constraints and develop a number of algorithms for implementing different ordering constraints. We show that our model allows to define an ordering constraint on each update operation, and the ordering implementation takes into account of detailed inter-operation semantics denoted by commutative operations and causal operations to reduce unnecessary delay and results in a better response time for update requests.

Key Words: Replication, Data consistency, Update ordering, Distributed databases, Performance evaluation.

1 Introduction

Distributed replication provides high availability, fault-tolerance and enhanced performance. But these features come at a price: replication adds great complexity to the system development [11, 6]. The most of all, replication jeopardises data consistency. In turn mechanisms have to be employed to enforce the data consistency. Maintaining the data consistency is very expensive, a common practice is then to relax the data consistency level as low as possible to give rise to better system performance.

Update ordering is an alternative data consistency model which has weaker semantics than that of the one-copy serialisability. The basic idea of the update-ordering model is to let replicas execute the same set of update requests in a sensible order. This order meets the requirements of both the clients and the data semantics of a replicated service application. Compared to the data replication in the transactional context, the update ordering model generally gives a better response time and a higher system throughput rate because it allows updates to be executed concurrently at different replicas in different orders.

We propose to use a set of ordering constraints to express the corresponding set of operations provided by a replica group. Each replica will execute the same set of update operations in a sensible order which is confined to the set of ordering constraints but may be different at replicas. In that case, we can achieve an improved system efficiency and throughput and still preserve the safety and correctness property of the system.

We organise our paper as follows. In Section 2, we develop the model for update orderings and constraints. In Section 3 we discuss the algorithms used to implement different ordering constraints. In Section 4, we carry out a performance study for our update ordering model. In Section 5 we discuss some related work and summarise the paper.

2 The Model of Update Orderings

In general, ordering constraints can be categorised into four types: FIFO, causal, total and total-causal [3] to reflect different semantical requirements of the replicated system and its clients.

A request message received by a replica \( R_i \) directly from its client is said to be originated from \( R_i \). We also need to distinguish a received request from a deliverable request. When a request is received by a replica, it is stored in a buffer/log and awaits to be checked on its ordering constraint. Once its ordering constraint is satisfied, that request is executable or deliverable, in other words, that request is ready to be executed by the replica.

Assuming an \( n \)-replica system, \( \mathcal{R}^n = \{ R_1, \ldots, R_n \} \), we define update ordering constraints as follows:

Definition 1: FIFO ordering constraint "\( \rightarrow \)". If two updates \( u_1 \) and \( u_2 \) originated and sent to the replica group from the same replica \( R_i \), and if \( u_1 \) is delivered before \( u_2 \) at the original replica, then \( u_1 \rightarrow u_2 \), iff \( u_1 \) is delivered before \( u_2 \) at the rest of replicas. §

Definition 2: Causal ordering constraint "\( \prec \)". If \( R_i \) delivered an update \( u_1 \) originated from \( R_j \) before sending out an update \( u_2 \), then \( u_1 \prec u_2 \), iff \( u_1 \) is delivered before \( u_2 \) at all replicas. §
Definition 3: Total ordering constraint "\(+\)". For two updates \(u_1\) and \(u_2\) sent from \(R_i\) and \(R_j\), then \(u_1 \leftrightarrow u_2\), iff: when one replica delivers \(u_1\) before \(u_2\), the rest of replicas deliver \(u_1\) before \(u_2\) as well; or the other way around, when one replica delivers \(u_2\) before \(u_1\), the rest of replicas deliver \(u_2\) before \(u_1\) as well. §

Definition 4: Total-causal ordering constraint "\(<\)". If two updates \(u_1\) and \(u_2\) are originated from \(R_i\) and \(R_j\) respectively, then \(u_1 \rightarrow u_2\), iff: \(u_1 < u_2\) and \(u_1 \leftrightarrow u_2\). §

FIFO is understood as requests sent by the same client are to be executed in the order they are sent group wide. Causal ordering is a natural extension to FIFO by considering different senders and it came from happened-before relation defined in [8]. Total ordering requires \(u_1\) and \(u_2\) to be delivered either in the order of \((u_1, u_2)\) or \((u_2, u_1)\), as long as the ordering is consistent at all replicas. The total-causal ordering is the integration of total and causal orderings. The strength levels of these ordering constraints are as follows: The total-causal ordering is the strictest and the free ordering (ie, no ordering) is the weakest.

To decide the ordering constraint for each update operation, we need to analyse the inter-operation semantics between update operations. The semantics is based on whether two update operations are commutative or not. Suppose that a server provides a set of update operations, \(OP_{srv} = \{u_1, u_2, \ldots, u_n\}\), and assume that \(u_1\) and \(u_2\) are any two update operations, \(u_1\) and \(u_2\) can be the same operation, we define their inter-operation semantics to have the following two relations:

Definition 5: Commutative relation "\(\parallel\)". \(u_1 \parallel u_2\), iff: the effect of executing \((u_1, u_2)\) equals the effect of executing \((u_2, u_1)\). §

Definition 6: Conflicting relation "\(\triangleright\)". \(u_1 \triangleright u_2\), iff: the effect of executing \((u_1, u_2)\) is different from that of executing \((u_2, u_1)\). §

It is easy to see that the conflicting relation is opposed to the commutative relation. Subsequently, we can give definitions on commutative operation and total operation.

Definition 7: Commutative operation. An update operation \(u\) is a commutative operation, iff: \(\forall v \in OP_{srv}, v \parallel u\). §

This is to say, if \(u\) is commutative with every operation in the \(OP_{srv}\), \(u\) is a commutative operation. A commutative operation implies that the order of its execution does not affect the state of replicas.

Definition 8: Total operation. An update operation \(u\) is a total operation, iff: \(u \in OP_{srv}\) and \(\exists v \in OP_{srv}, v \triangleright u\). §

This is to say, if \(u\) is conflicting with one operation in \(OP_{srv}\), \(u\) is a total operation. When an update operation is a total operation, it implies that the total ordering constraint has to be applied. This is because if such an operation is submitted to a replica, its conflicting operation may be issued at a different replica concurrently. If they are executed in different orders, the state of replicas will be different too, which means data consistency is violated. In other words total operations have to be executed sequentially at all replicas to maintain the data consistency.

The causal ordering based on the happened-before semantics captures the potential cause-effect relation between two update events. But these two events can be totally unrelated from the data semantics point of view. Here we define the real causal relation that captures the semantical causal effect relation, apart from just having the happened-before in a distributed environment.

Definition 9: Real causal relation "\(\lt\)". If \(u_1 \lt u_2\), iff \(u_1 < u_2\) and \(u_1 \rightarrow u_2\) is the real effect of executing \(u_1\). §

Definition 10: Real causal operation. An update operation \(u\) is a real causal operation, iff: \(\exists v \in OP_{srv}, v \lt u\). §

Therefore, we can define the following three types of operation sets.

Definition 11: Total operation set - Total_{op}. Total_{op} contains all total operations out of \(OP_{srv}\). §

Definition 12: Commutative operation set - Comm_{op}. Comm_{op} contains all commutative operations out of \(OP_{srv}\). §

Definition 13: Causal operation set - Causal_{op}. Causal_{op} contains all real causal operations out of \(OP_{srv}\). §

Now we can derive an assertion:

Assertion 1. Comm_{op} \cup Total_{op} = OP_{srv} and Comm_{op} \cap Total_{op} = \emptyset.

Proof. This is equivalent to prove that, \(\forall u \in OP_{srv}\), either \(u \in Comm_{op}\) or \(u \in Total_{op}\), but not \(u \in Comm_{op}\) and \(u \in Total_{op}\).

According to Definition 7 and Definition 8, if \(u\) is a commutative operation (in Comm_{op}), \(u\) is commutative with every operation in \(OP_{srv}\), thus \(u\) is not in Total_{op}. Also if \(u\) is a total operation (in Total_{op}), \(u\) is conflicting with at least one operation in \(OP_{srv}\), hence, \(u\) is not a commutative operation (not in Comm_{op}). §

Generally, we can use a two-dimensional matrix to represent the commutative relation between any two operations. Then Total_{op} and Comm_{op} can be easily derived from the matrix.
Definition 14: Commutative matrix -
\[ \text{CommMatrix}[u_1, u_2][u_1, u_3] = \begin{cases} 
1 & \text{if } u_1 \parallel u_2, \\
0 & \text{if } u_1 \equiv u_2.
\end{cases} \]

The total operation set and commutative operation set can be derived by:
\[ \text{Total}_{op} = \{ u_i, u_j \in OP_{srv} \mid \exists u_j \in OP_{srv} \land \text{CommMatrix}[u_i, u_j] = 0 \} ; \]
\[ \text{Comm}_{op} = OP_{srv} - \text{Total}_{op} . \]

Definition 15: Causality matrix -
\[ \text{CausalMatrix}[u_1, u_2][u_1, u_3] = \begin{cases} 
1 & \text{if } u_i \prec u_j, \\
0 & \text{otherwise}.
\end{cases} \]

The causal operation set can be derived by:
\[ \text{Causal}_{op} = \{ u_i, u_j \in OP_{srv} \mid \exists u_j \in OP_{srv} \land \text{CausalMatrix}[u_i, u_j] = 1 \} . \]

3 Implementations of Ordering Constraints

FIFO is often well supported by underlying communication primitives provided by the operating system, such as TCP/IP reliable streams. TCP/IP protocol guarantees that messages transmitted to the destination in the sending order. Otherwise, each replica maintains a message counter dispatching a sequence number to each update request sent out. Subsequently, those update operations originated from the same replica can be executed at peer replicas in the same order as of their original replica simply by respecting their sequence numbers.

General implementation for causal ordering is by using the Vector Timestamp protocol [5, 13], which is described by Listing 3.1. For an n-replica group \( R^n = \{ R_1, \ldots, R_n \} \), a vector timestamp \( VT_{R_i} \) is created and maintained by the replica \( R_i \) at its local space, where \( VT_{R_i} = VT[1..n] \). The basic idea of this protocol is to let each update request carry a vector timestamp (VT) representing its causality, this causality is checked for deliverability at remote sites. The following is the full protocol algorithm.

Listing 3.1 — Vector Timestamp Protocol

At \( R_i \), where \( 1 \leq i \leq n \), \( VT_{R_i} \) is initialised as \( VT_{R_i}[1..n] := [0, \ldots, 0] \).

\( VT_{R_i} \) is updated upon the following events:

Rule 1: Assigning VT. When an update \( u \) is originated from \( R_i \):
\[ VT_{R_i}[i] := VT_{R_i}[i] + 1 \]
When \( R_i \) propagates this message to peer replicas, the message carries the current value of \( VT_{R_i} \).

Rule 2: Checking the deliverability of a causal ordering operation.
If an update \( u \) from \( R_j \) carrying \( VT_{R_j} \) is received at \( R_i \):
\[ \text{if } (VT_{R_j}[k] \leq VT_{R_i}[k], 1 \leq k \leq n \text{ and } k \neq j) \{ \]
\[ \text{u is deliverable;} \]
\[ VT_{R_i}[j] := VT_{R_i}[j] + 1 ; \]
\[ \text{else u is deferred}; \]

If we only consider real causal operations (i.e., operations in \( \text{Causal}_{op} \)), we can improve the vector timestamp protocol by modifying the Rule 2 of Listing 3.1 as:

Listing 3.2 — Improved Vector Timestamp Protocol

Rule 2: Checking the deliverability of a causal ordering operation.
When an update \( u \) from \( R_j \) carrying \( VT_{R_j} \) is received at \( R_i \):
\[ \text{if } (u \in \text{Causal}_{op}) \]
\[ \text{if } (VT_{R_j}[k] \leq VT_{R_i}[k], \text{where } 1 \leq k \leq n \text{ and } k \neq j) \{ \]
\[ \text{u is deliverable;} \]
\[ VT_{R_i}[j] := VT_{R_i}[j] + 1 ; \]
\[ \text{else m is deferred}; \]
\[ \text{else } \{ \# u \notin \text{Causal}_{op} \}
\[ \text{u is deliverable;} \]
\[ VT_{R_i}[j] := VT_{R_i}[j] + 1 \}

This means if receiving a remote operation that is in \( \text{Causal}_{op} \), its causality is checked, otherwise, its causality is not checked. An operation not in \( \text{Causal}_{op} \) is a causally free operation which means it has no cause-effect relation with any other operation in the operation set. The saving of this improved vector timestamp protocol is on checking those real causal operations instead of all operations in the operation set. This will speed up the number of requests being handled by the system.

We use a simple approach — the centralised sequencer for implementing the total ordering of update requests. To be able to decide a total-ordering operation is deliverable, each member keeps a variable of \( \text{USN.major} \) in its local space to record the maximum USN executed so far. If a total-ordering operation arrived holds the next unique sequence number (USN), then this operation is ready to be executed. Otherwise the operation is deferred until lower USN operations are performed. Here we give the full algorithm for the centralised sequencer method.

Listing 3.3 — Assigning the USN Protocol

Rule 1: Acquiring a USN from the sequencer.
At each replica site:
  while (true) {
     receive (client, u); // u is a total-ordering update.
     send (the.sequencer, USN-request);
     receive (the.sequencer, USN-reply);
     u.USN := USN-reply.USN;
     multicast (u);
  }

Rule 2: Assigning the USN to a member's request.

At the sequencer site:
  int USN.counter := 0;
  while (true) {
     receive (member, USN-request);
     USN.counter := USN.counter+1;
     USN-reply := USN.counter;
     send (the.member, USN-reply);
  }

Rule 3: Checking if a total-update operation is executable.

At each replica site:
  int USN.major := 0;
  while (true) {
     receive (member, u);
     if u.USN == USN-major +1 
       u is executable;
     else u is deferred;
  }

Commutative operations can be executed right away at local replicas, since their ordering does not affect the final state of replicas as long as they are propagated to other replicas eventually.

Now we give an improved version for implementing the unique sequence number generator by using the knowledge represented by the commutative matrix CommMatrix. According to Definition 8, an operation u is a total operation may not conflict with every operation in OPu. So if two operations u1, u2 ∈ Totalop and u1 ∥ u2 received at the sequencer consecutively for USN, the same USN will be given to both of them, so that u1 and u2 can be executed concurrently at their original replicas without u2 being deferred for the arrival of u1.

The improved USN protocol assigns each update operation a USN which contains two fields, the USN-major and the USN-minor. If the sequencer receives a sequence of total operations which are commutative, i.e. any two of them are commutative in the sequence. The USN-major and the USN-minor are assigned the same value for all of them in the sequence. Thus, when a replica receives any update operation from this sequence, they can be executed right away without being deferred, whereby if using the algorithm of Listing 3.3, this sequence of operations would have been executed in the ordering received by the sequencer at all replicas. The detailed protocol is as follows:

Listing 3.4 — The USN Protocol Improved by Considering Commutative Pairs

Rule 1: Acquiring a USN from the sequencer.

At each replica site:
  while (true) {
     receive (client, u); // u is a total-ordering operation.
     send (the.sequencer, u, USN-request);
     receive (the.sequencer, u, USN-reply);
     u.USN := USN-reply;
     multicast (u);
  }

Rule 2: Assigning the USN to a member's request.

At the sequencer site:
  CommOP.set := {}; // hold commutative operations
  int USN.major := 0; // counter for assigning the major value
  int USN.minor := 0; // counter for assigning the minor value
  int last.minor := 0; // number of commutative operations in the last sequence
  while (true) {
     receive (member, u, USN-request);
     if CommOP.set == {} {
       USN.major := USN.major+1;
       USN.minor := 1;
       CommOP.set := {u};
     } else if (u || v, where v ∈ CommOP.set) {
       CommOP.set := CommOP.set + {u};
       USN.minor := USN.minor+1;
     } else {
       CommOP.set := {u};
       USN.major := USN.major+1;
       last.minor := USN.minor;
       USN.minor := 1;
     }
     USN-reply := (USN.major, last.minor);
     send (the.member, USN-reply);
  }

Rule 3: Checking if a total-update operation is executable.

At each replica site:
  int t.major := 0;
  int t.minor := 0;
  while (true) {
     receive (member, u);
     if (u.major == t.major) {
       t.minor++; // u is deliverable;
     } else if (u.major == t.major+1) and (u.minor == t.minor) {
       t.major++; // u is deliverable;
     } else if (u.major == t.major+1) and (u.minor > t.minor) {
An operation is a total operation (in Total$_{op}$) and also a causal operation (in Causal$_{op}$) has the total+causal ordering constraint. The implementation for total+causal operations can be done by combining the vector timestamp protocol and the USN protocol (Listing 3.1 and Listing 3.3). Each total+causal update propagated to the group carries a Time Stamp (TS) which consists of two fields: TS := (VT,USN). The update is deliverable if VT and USN are both satisfied.

We derive the second assertion here:

**Assertion 2.** If Total$_{op}$ := OP$_{rv}$, the centralised USN protocol (Listing 3.3 and 3.4) guarantees causal ordering as well.

**Proof.** The assertion states that, for two updates u1 and u2, if u2 causally depends on u1 at one replica, the USN protocol guarantees that u1 will be delivered before u2 at all other replicas.

Because of Total$_{op}$ = OP$_{rv}$, this means all operations are total-order operations. If u2 causally depends on u1, this means u1 is delivered at one replica before u2, thus u1 holds an earlier USN, and u2 holds a later USN. Since u1 and u2 are both total-order operations, therefore, under the USN protocol, u1 will be delivered before u2 at all replicas.

This means if Total$_{op}$ = OP$_{rv}$, there is no need to specify total + causal constraint to any operation. The implementation can be optimised to attach only USN to each update propagated to the group, the vector timestamp is not needed.

To find the least strict ordering constraint for each update xation of OP$_{rv}$, we need to analyse the commutative relation and causal-effect relation between each pair of update operations, in other words, to construct the CommMat and CausalMat. Then total operation set (Total$_{op}$) and causal operation set (Causal$_{op}$) can be derived from the two matrices. If an operation u does not belong Total$_{op}$, nor Causal$_{op}$, it is a commutative operation.

Commutative operation is constrained by the FIFO constraint. If u belongs to both Total$_{op}$ and Causal$_{op}$, u has to be associated with total+causal constraint. If Total$_{op}$ is empty, the commutative matrix can be used to represent commutative pairs among total operations.

A causal operation carries the vector timestamp (VT); a total operation carries the unique sequence number (USN); and a total+causal operation carries the (VT,USN) as we explained in the previous section. Besides, all update requests carry a sender identifier which is formed by using main identifier, client sequence number, replica identifier, and the sequence number dispatched by the replica. The client identifier and the client sequence number tells the origin of the request and its order at the origin so that the requests from the same client can be executed sequentially. The replica identifier and the replica sequence number is used to enforce that the order of requests being executed at original replica is kept at peer replicas.

### 4 Performance Evaluation

The evaluation is based on two metrics: the average response time (ART) over requests, and the average system throughput (AST) which is the average number of requests that can be executed by the system.

Assume we have an n-replica system: R, with R = {R1, . . . , Rn}. To evaluate the overall ART of the replication system, assuming the ART at each replica is ARTRi, where 1 ≤ i ≤ n, then we define the overall ARTR to be the average value of average response times achieved at each replica:

\[
ARTR = \frac{1}{n} \sum_{i=1}^{n} ARTR_i.
\]

To evaluate the overall AST, we assume that N requests are received by each replica of the group. The replicated system starts from a global consistent state. By executing all n·N at all replicas, the replicated service system stops at another globally consistent state. Let ASTRi represent the AST at Ri; it is measured by the average number of requests executed at Ri to finish all n · N requests, then we can define the overall ASTR to the minimum value among all ASTs achieved by each replica:

\[
ASTR = \min(\text{ASTR}_1, \ldots, \text{ASTR}_n).
\]

That is, the overall system throughput is determined by the replica who is the last one to finish executing all n·N requests. However, these two metrics can be largely affected by many factors, such as, the replication degree, the number of clients connected to the replicated system, the update percentage, the propagation frequency, etc. In this paper, we concentrate on the study of the ART and the AST with respect to propagation frequencies, but fixing the replication degree to four replicas and update percentage to 100%.

In our experiments, we allocated four Sun Sparc stations, named "bofur", "bifur", "durin" and "elwing", respectively, for running four replicas. These workstations are connected by a local 10Mbps Ethernet. The programming environment is Java 1.2 and Sun Solaris 5.6. The most involved Java packages are Java network (java.net) and Java input/output (java.io) packages. Since each of the
workstations has a slightly different hardware configuration from others, in turn each machine has shown a slightly different performance.

We use a service application which is composed of eight different update operations, represented by index numbers from 0 through 7 inclusive. Each replica has a child thread simulating a client that issues requests one after another. In the following experiments, each client thread sends a sequence of 100 requests, which is generated by a random function that produces well-balanced numbers (between 0 and 7 inclusive) to simulate the sequence of operations in the real environment.

Thus we have \( n = 4, N = 100 \). The ART at each replica is evaluated by the average response time over 100 requests in millisecond. The AST is measured by the average number of requests executed per second at each replica by finishing 400 requests (100 requests are issued by its client, and 300 are propagated from peer replicas).

![The impact of ARTs](image)

**Figure 1.** The ARTs at Different Propagation Frequencies

![The impact of ASTs](image)

**Figure 2.** The ASTs at Different Propagation Frequencies

This experiment is to study how the propagation frequency affects the response time and the system performance. Propagation frequency affects a system where most of operations are commutative or causal. This is because if an operation is a total or total-causal operation, the propagation is triggered right away. In other words, total or total-causal operations break up the regularity of propagations at the specified frequency.

In this particular experiment, we assume that all eight operations are commutative operations so that the propagations happen exactly at the frequency specified. We let the propagation frequency vary at: (1) every 1 request; (2) every 2 requests; (3) every 5 requests; (4) every 10 requests; (5) every 20 requests; (6) every 50 requests.

Figure 1 shows the testing results of ARTs over 100 requests at varying propagation frequency rates on four machines. From the figure, we can observe that at the frequency of every 5 and 10 requests, the ARTs are at the lowest level (the best). When propagation happens more frequently, i.e., less than every 5 requests, the ARTs tend to be higher. When the propagation happens less frequently, i.e., every more than 10 requests, the ARTs tend to grow slightly. This can be explained by the fact that the ART is determined by whoever becomes the prominent factor, either the more frequent small delays or less frequent large delays.

Figure 2 shows what the ASTs are achieved on four machines. At the frequency of every 5 and 10 requests, replicas have relatively high throughput rates. This matches the test results depicted by Figure 1, where the ARTs at the frequency of 5 or 10 are the lowest, in turn, highest throughput rates should be achieved there.

We also conducted experiments related to the semantics of update operations, these results are included in a full paper. Some of our findings are listed below:

- Identifying commutative operations and reducing the strength level of each operation's ordering constraint can improve the system performance significantly.

- Identifying commutative operation pairs and real causal operations improve the system performance slightly.

- Decreasing the propagation frequency does not always improve the overall performance. The performance reaches the best at some frequencies (best frequencies). Then the performance starts to worsen when the frequency values fall out of those best frequencies.

5 Related Work and Summary

Data consistency based on update ordering is a weaker consistency model than the one-copy serialisability model. Many update protocols for replicated objects have been developed to explore the rich semantics of update operations [17, 15, 12, 16]. For example, the lazy replication [7, 9, 4] is an example of using asynchronous propagation technique.
in replication systems. Another form of a lazy replication is the Local Consistency (LC) approach [1] in which replicas that update shared data object are not responsible for informing other replicas about the update; instead, a replica ensures that the object copies it accesses are consistent.

The implementation of lazy replication uses a timestamp label on each operation (both query and update). The timestamp is similar to the vector timestamp method described in this paper. The label determines the ordering constraint that has to be satisfied before the query or update can be executed.

The lazy propagation of updates from one replica to other replicas saves underlying network communication cost. However, the lazy replication system makes no use of process group concept. No attention has been made about preserving consistent state upon replica membership changes. Gossip messages are sent to other replicas one by one.

Group communication systems such as Horus [14] and Totem [10, 2], provide reliable and ordered multicasting primitives for building group-oriented distributed systems. These projects differ in the properties of the ordered multicast services provided, and in their assumptions about the underlying communication delays and failure semantics.

Building a replication system based on the group communication systems has to rely on multicasting primitives to ensure the data consistency among replicas of the group. Multicasting primitives provided by the group communication systems multicast requests on a one-to-one basis. They do not provide the facility of packing multiple requests into one message and unpacking the message at remote sites.

Our update-ordering data consistency captures the advances of group communication mechanisms and the lazy propagation. It can be seen as an improved version to lazy replication by deploying group communication concepts. Based on the update ordering and data consistency modelling, we can build a replication system that has the flexibility of choosing a different ordering constraint for each update operation with respect to the data semantics of the replicated service application to maximise the system efficiency.

References


The Design and Implementation of a Model for Database Publishing on the WWW

WanLei Zhou
School of Computing and Mathematics, Deakin University, Clayton, Victoria 3168, Australia
HaiYi Zhang
Department of Computer Science, Harbin Institute of Technology, Heilongjiang, 150001, China

Abstract

The paper studies database publishing technologies based on the WWW. The dramatic progress of the Internet, especially the WWW, makes it possible for people to share database information in a very large scale. Database publishing is an important application in this area and it involves the key technologies, such as the HTTP protocol, server technology, client browser technology, markup language HTML, etc. Typically, there are two means to publish databases on the web: CGI-based and Java-Based. This paper analyzes the advantages and disadvantages of the two methods. Then, it puts forward an approach to database publishing based on a new model: the extension of HTTP. The result of a preliminary evaluation shows that the new model is an efficient approach.

Keywords: WWW, Database Publishing, CGI, Java, Client/Server Computing, Database Connectivity.

1 Database publishing based on WWW

Database publishing means providing database information owned by an individual or an organization to other people to access [1, 4]. The publishing range can be local publishing or remote (distributed) publishing [17]. There are two kinds of interface for general databases: local interface and client/server interface [8], and database publishing must pass through database interfaces. In the environment of the World Wide Web, people are most interested in distributed (or client/server) database publishing [12].

The existing database publishing means can be classified into two types: CGI-based and Java (JDBC)-based. We analyze the advantages and disadvantages of the two methods here.

1.1 Common Gateway Interface (CGI)
CGI is one of the early and still most widely used techniques for integrating databases into the Web environment. It is the specification for transferring information between a Web server and a CGI program (script). Running a CGI script from a Web browser is mostly transparent to the user. However, several things must occur for a CGI script to execute successfully [3]:

Information can be passed from the browser to the CGI script in a variety of ways, including: passing parameters on the command line, passing environment variables to the CGI programs, passing data to CGI programs via standard input, and using extra path information. The script can return the results with embedded HTML tags, as plain text, or as an image. The browser then interprets the results like other document. This provides a very useful mechanism permitting access to any external databases that have a programming interface.

The main advantages of CGI are its simplicity and language independence, Web server independence, and its wide acceptance. Despite these advantages, the CGI-approach does have some problems. The first problem is that the communication between a client and the database server must always go through the Web server. This may possibly cause a bottleneck if there are a large number of users accessing the Web server simultaneously. For every request that is exchanged, the Web server has to convert data from or to an HTML document. This necessary conversion process adds significant overhead when processing a database query.

The second problem is the lack of efficiency and transaction support in a CGI-based approach. Every query that is submitted through CGI is treated as the first encounter. This, however, is due to the statelessness of HTTP. As a result, the database server has to perform the same logon and logout procedure, even if the same user submits multiple queries. The CGI script could attempt to overcome this problem by handling queries in a batch mode, but then transactions that required multiple interactive queries would be difficult.

Another important disadvantage is that the server has to generate a new process or thread for each CGI script. For a popular site that can easily acquire dozens of hits almost simultaneously, this can be a significant overhead, as the processes would compete for memory, disk, and processor time. The script developer may have to take into consideration that there may be more than one copy for the script executing at the same time, and consequently have to allow for concurrent access to data files used.

The final disadvantage of CGI is security. If appropriate measures are not taken, security can be a serious threat in CGI applications. Many of these security problems relate to the data that is input by the user at the browser end. If the user
input is not checked and handled correctly by the script, dangerous security breaches could occur.

1.2 Java Database Connectivity (JDBC)

The JDBC approach [9] is another popular implementation for database publishing. The JDBC package is a set of Java classes that defines a database access API (application programming interface). It supports basic SQL functionality and it enables access to a wide range of relational DBMS products. In using JDBC, Java can be used as the host language for writing applications for database publishing.

The advantage of using JDBC drivers that make use of ODBC is that they are readily available for many of the most popular DBMSs for a very low price. The Java-based approach allows a connection between a client and a server to be set up during the cause of the communication, and therefore eliminates the need of multiple database connection operations when the user submits multiple queries. However, there are disadvantages with the Java-based approach:

- A JDBC driver that is not a pure Java implementation will not necessarily work with a Web browser.
- For security reasons, currently an applet that has been downloaded from the Internet can connect only to a database located on the host machine from which the applet originated. This is part of the sandbox concept designed into Java applets.
- Deployment costs increases with the need to install, administer, and maintain a set of drivers, or database software for each client system.

The major advantages of using JDBC are the cross-platform independence and the ability of delivering database functionality using Java applets through the Internet. The JDBC package specifies the interfaces between Java and databases (based on SQL2). JavaSoft (the group that defined JDBC) only specifies the interfaces. All implementation of JDBC drivers is done by third part companies with special expertise. A change of a driver will not change the program. However, since Java is an interpreting language, the efficiency of the program execution is rather low because the interpreting needs time.

After the consideration for the two publishing means, we propose a new database publishing method that is based on an extension of HTTP protocol.

2 The model of database publishing based on an extension of HTTP protocol

2.1 The model
As the progressing of the Internet, especially the WWW, the relationship between database publishing and Web server—HTTPD (HTTP Daemon) becomes more and more close [5, 6]. Therefore it seems to be a good strategy for the HTTPD to integrate the database functionality [2]. Here, we propose a model of database publishing based on an extension of HTTP protocol, as shown in Figure 1.

In the browser, we use the Netscape plug-in API to develop a database plug-in as the front end for data access. The plug-in is corresponding to the non-enroll MIME type which defines as database/x-SQL ([7, 11]). Its extended name is defined as db.

![Figure 1 A publishing model based on the extended HTTP protocol](Image)

As soon as the browser meets MIME type that is data of database/x-SQL, it will call the database plug-in to deal with the data. Comparing to the publishing based on CGI, this publishing method is powerfully interactive. Its weakness is that when accessing the publishing database, it has to use the plug-in in the front end.

When the plug-in communicates with HTTP, it uses the extended HTTP protocol. In order to perform the search function, we extended a method called DATABASE for the method set of the HTTP/1.0 protocol [14] and defined request/response messages. But we still keep the statelessness property of the HTTP protocol. In the server end, we have several modules for different requests. After the HTTPD receives a request message, it can call a module related to the request method. Here, for the safety reasons, the database engine only provides searching function. It has no adding, deleting and modifying functions.

### 2.2 The management of the result set and history records

If the server sends the searching results to the front end every time and if the amount of data is huge, it will take a longer respond time, and the user will need to search some subset of the result set. Obviously, this approach will cost unnecessary time. So, we require that the database engine only return a fixed number of records for each searching. Since all the protocols in use are stateless, we create the buffer-
mechanism and history management in order to enhance the efficiency of the publishing.

The statelessness property of the extended protocol means that, if a buffer mechanism for searching result set is not created in the client end, in order to obtain the data the browser must interact with the server for every time in backward or forward browsing. That practice wastes time and therefore the efficiency is low. Here, we use the FIFO algorithm for result management at the client end. The data structure for the algorithm implementation is an FIFO queue of a fixed length. The searching respond message returned from the server end is sent to the queue. If necessary, the result item that is stored first is eliminated and it will be sent to the history stack (for the browser to use later).

We use the LRU algorithm for History record management. A result set is corresponding to a triple (query statement, start record number, record number). In fact, every entry of the history record stack is a triple corresponding to a result item.

3 Description of the extended protocol

We define a database message to describe the extended database access protocol based on the HTTP protocol. Refer to RFC1945 [14] for a more detailed description on the HTTP protocol. Here, the message pattern is in BNF and its general structure is defined by the Internet Mail formats [13].

(1) Meta-request message
Pattern: DATABASE database name HTTP-version CRLF
* (general-Header | Request-header | Entity-Header) CRLF
[Entity-Body]

When HTTPD receives the message, it can distinguish which database model that should be called to deal with the request from the DATABASE method. According to the request's database name, the database model interacts with the embedded database engine and receives meta-information of requested database, such as tables in database and attributes of individual area in the table. The database model packs the meta-information in the entity of meta-information response text, and sends to the browser end plug-in. The user can reference to those information for search operations.

(2) Meta-information message
Pattern: HTTP-version Status-code Reason-Phrase CRLF
* (General-Header | Response-Header | Entity-Header) CRLF [Entity-Body]

Here, the Status-Code points out that the request is successful or not. If successful, the Entity-Body gives the meta-information of requested database. Otherwise, the Reason-Phrase gives the reason of the failure, for instance, undiscovered database, illegal operation, etc. and the Entity-Body gives the detailed information about failure.
(3) Search request message

Pattern: DATABASE database name HTTP-version CRLF
* (General-Header | Request-Header | Entity-Header) CRLF [Entity-Body]

Here, Entity-Body has a pattern as follows:
[Start-Record: begin number of record CRLF]
Record-Number: record number of request CRLF
Query-statement

After the HTTPD receives the message, it can perform a searching operation. If this is a new searching, the head of the Start-Record might be omitted. The Record-Number stands for the record number for the current request. Following a request message, the database model can return the result. The Start-Record and Record-Number will divide the searching result of Query-Statement into several subsets and return a subset for each request. Here, a search request of the client must have a Query-Statement.

(4) Search respond message

Its pattern is the same as the search request message pattern. The only difference is the Entity-Body pattern. There are two kinds of Entity-Body patterns for a search respond message:

- **No Start-Record header** means for this time the searching statement of the request is different from that of the last time for the client end. The HTTPD can return the first Record-Number records of the result set. The pattern of the Entity-Body is:

  Total-RecordNumber : RecordNumber-In-This-Message : Meta-Information : Records

  Here, the Total-RecordNumber stands for the total record numbers of this searching. The RecordNumber-In-This-Message stands for the record numbers included in this message. The Meta-Information stands for the meta information that is included in the returning result set. The Records is the data of the result set.

- **With a Start-Record header**, HTTPD will return records from the result set starting from the Start-Record and returns the number of Record-Number records. The Entity-Body only includes the record set of the request.

4. The efficiency evaluation

In order to fairly evaluate the three database publishing methods: the CGI-Based, the Java-Based and the Extended HTTP protocol, we carried out some preliminary experiments using the MySQL database system [10] as the database engine. The result of our experiments indicated that in general, the database publishing method
we proposed is the most efficient approach, the Java-based method is in the middle, and the CGI-Based method is the least efficient.

4.1 The environment and design for the experiments

The testing environment has the following components,

1. Database system for testing: mSQL database, and the size is about 350 records.
2. The configuration of computer for mSQL server execution: CPU-80586 100, operating system - Linux 1.2.3, memory—40 Mb.
3. The browser for testing is Netscape Navigator 3.0. The configuration of computer for running the client is: CPU—80586 100, operating system – NT4.0, memory – 32 Mb.

In order to guarantee accuracy, we set the following conditions during testing:

1. Absolutely quiet for network, no network noise.
2. Same client computer and server computer.
3. Running on the same Netscape Navigator Browser at the client end and using the same mSQL engine.

Table 1 lists the testing front end for every method:

<table>
<thead>
<tr>
<th>Publishing Means</th>
<th>Testing front end</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on Java</td>
<td>Java Applet</td>
</tr>
<tr>
<td>Based on CGI</td>
<td>Java Applet</td>
</tr>
<tr>
<td>Extended HTTP Protocol</td>
<td>Netscape Plug-in</td>
</tr>
</tbody>
</table>

Table 1. Testing end of the publishing methods

Because the testing front end of the CGI-based method uses standard HTML pages, therefore it is impossible to test its respond time. Here we use Java Applet to test the front end. It simulates the browser to request CGI programs. We believe that using this method to test respond time should be the same as using a browser directly, because an applet running in a browser is an independent program, and the time for evaluating efficiency does not include processing time for the browser to process query results. In addition, we need to point out that the testing front end for the extended HTTP protocol simply is to (1) create the connection to HTTPD, (2) send query respond message, (3) receive query results and (4) close the connection. It does not process display of results (there is no GUI interface).

The testing procedures for the three testing front ends are listed in Table 2.
<table>
<thead>
<tr>
<th>Publishing Means</th>
<th>Testing procedure for Front End</th>
</tr>
</thead>
<tbody>
<tr>
<td>Based on Java</td>
<td>Connect to mysql; Select database; Query; Close the connection</td>
</tr>
<tr>
<td>Based on CGI</td>
<td>Connect to HTTPD; Send CGI request; Accept respond; Close HTTP connection</td>
</tr>
<tr>
<td>Expanded HTTP Protocol</td>
<td>Connect to HTTPD; Send query request; Accept respond; Close HTTP connection</td>
</tr>
</tbody>
</table>

Table 2. The testing procedure for front ends

Here, wherever there is an interaction between the server end and the front end, the time spent in the interaction will be sampled. For example, the time sampling for the CGI-based method is:

Time sampling 1: create connection to HTTPD; time sampling 2; send CGI request; time sampling 3; accept respond, time sampling 4; close connection; time sampling 5; testing completed; obtaining time result for the operation.

4.2 The experiment process and analysis of results

Here, we use four query statements to test the respond time for all database publishing methods. The four query statements are: simple query, general query (conditional query), complex query and query with external connection. For every query statement of every database publishing method, we tested five groups of data and take an average number as the test result. Table 3 presents the testing results. Where each query statement is listed below:

Statement 1: SELECT DISTINCT * FROM student
Statement 2: SELECT DISTINCT * FROM student WHERE age = 22
Statement 3: SELECT DISTINCT * FROM student WHERE age = 22 ORDER BY name
Statement 4: SELECT DISTINCT student.name, student.sex, student.age, student.phone, list sdflhg FROM student, list WHERE student.name = list dsghfl ORDER BY student.name

<table>
<thead>
<tr>
<th>Method Interactive Time</th>
<th>Statement</th>
<th>Base on CGI</th>
<th>Based on Java</th>
<th>Extending HTTP</th>
<th>Number of records returned</th>
</tr>
</thead>
<tbody>
<tr>
<td>Statement 1</td>
<td>1080</td>
<td>1073</td>
<td>987</td>
<td>75</td>
<td></td>
</tr>
<tr>
<td>Statement 2</td>
<td>1083</td>
<td>1040</td>
<td>1004</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>Statement 3</td>
<td>1130</td>
<td>897</td>
<td>1016</td>
<td>69</td>
<td></td>
</tr>
<tr>
<td>Statement 4</td>
<td>1670</td>
<td>1634</td>
<td>1533</td>
<td>69</td>
<td></td>
</tr>
</tbody>
</table>

Table 3. Testing Results
For each method, its efficiency can be defined as:

\[ E = \frac{\text{The number of records returned for a single search}}{\text{Time of interaction for a single search (ms)}} \]

In order to compare the efficiency for different database publishing methods, we show the testing result in terms of \( E \) in Figure 2.

![Figure 2. The comparison of efficiency for the three publishing methods](image)

For every publishing method, its efficiency is the inverse ratio with respond time, but is a direct ratio with record numbers returned of a single interaction. Here, we need to pay attention to the fact that the testing result is a single interaction rather than repeated interactions. For database publishing based on Java, there is a connection between the client and the server and a following query does not repeat the starting procedure of the previous query. In such a case, its efficiency should be better than the method that based on the extended HTTP protocol.

From Figure 2, we can see that in general, the extended HTTP protocol is the best in terms of efficiency; the Java-based method is the second; then followed by the CGI-based method. The reasons why the Java-based method is better than the CGI-based method are that the Java-based method does not need to forward messages in the server end but directly interacts with the client; whereas the CGI-based method not only requires a starting time but also needs to forward messages for four times.

There are two reasons why the extended HTTP protocol method is better than the Java-based method. Firstly, Java execution is slow because of its interpretation. The extended HTTP protocol method also does not need a starting time and has no message forwarding. Secondly, when the Java front end interacts with the server, it
includes four times of network interaction, and therefore there are eight times of network transmission in total. In general, network transmission time is much higher than the process time for the server end. This is the main reason.

5 Remarks

The World Wide Web and the Internet have revolutionized the information technologies. Services are now provided through the WWW in almost any application areas like business, commerce, education, etc [16, 18]. The popularity of WWW makes it possible for users everywhere to access and process information either in a local network or remotely. Web-based computing allows organizations to integrate disparate systems, especially database systems into a single, coherent environment [15] and to publish database information to service both internal and external customers effectively. However, as normally the services for database publishing are accessed by a huge number of users, the efficiency of the method used for database publishing is crucial. This paper analyzed the advantages and disadvantages of the two most popular methods in database publishing, and proposed a new method based on an extension to the existing HTTP protocol. The paper then evaluated the three methods through experiments and concluded that the new method is more efficient than the two existing methods.

References

MANAGING MOBILE TRANSACTIONS

WANLEI ZHOU and JIANTING ZHONG
School of Computing and Mathematics, Deakin University
Geelong, VIC 3217, Australia
{wanlei, jtz}@deakin.edu.au

ABSTRACT This paper addresses the problem of managing transactions in a mobile and replicated environment. We first establish the transaction model. Based on this transaction model, we present the design of our system for managing transactions in the mobile and replicated server environment. Finally, we present an implementation example of the system.

Key Words: Mobile computing, Distributed systems, Transaction management; Replication; Distributed databases.

1 INTRODUCTION

Two types of computing units exist in a mobile computing environment: the static units and the mobile units [7]. A static unit never changes its location during the course of computing, while a mobile unit can always change its location, and can be disconnected from and then re-connected to the static units. To improve the reliability and availability, the data of a mobile computing system are always replicated.

A transaction in such a mobile and replicated environment (the transaction is called a mobile transaction [5]) is very difficult to manage due to the mobility of the mobile units and the replication of data in both mobile and static units. A mobile transaction can be initialized from any mobile or static units. The units involved in a mobile transaction can be located/replicated in any mobile and static units.

Transaction is one of the challenges that mobile computing offers for database research [2]. There have been some research on the problem of consistent access to replicated data in often-disconnected mobile computers [1]. The fundamental problem is that standard transactional approach to the propagation of updates to replicas is unstable [3]. Mobile applications require lazy replication algorithms that asynchronously propagate replica updates to other nodes after the updating transaction commits. The purpose of this paper is to propose an approach for managing transactions in a mobile computing environment with replicated servers.

2 THE SINGLE SERVICE MODEL

We start with a single service model (SSM) where a mobile computing system only provides one service. A service is provided by a group of replicated servers (called replicas) executing on some computing units.

We classify replicas of a service into two types: a primary replica and several non-primary replicas. A primary replica (naturally) is located in a static unit to increase the reliability. For simplicity, we assume that a primary replica never fails (an election algorithm [6] can be used to elect a new primary replica when the primary replica fails). A non-primary replica can be located in a static or a mobile unit, and each computing unit (static or mobile) executes one replica.

Formally speaking, the service $S$ is supported by a set of $k$ replicas: $S = \{R_1, R_2, \ldots, R_k\}$. Without loss of generality, we assume that $R_1$ is the primary replica for service $S$. Let $O^1 = \{d_1, d_2, \ldots, d_n\}$ be the data objects managed by primary replica $R_1$, then the consistency constraint requires that (eventually) for $i = 2, 3, \ldots, k$, $O^i \equiv O^1$, where $O^i$ is the data objects managed by non-primary replica $R_i$.

Each replica in our system provides a number of methods that can be called by clients for processing the data objects managed by the replica. We use $P$ to denote the set of all methods provided by all replicas of the system:

$$P = \{p | p \text{ is a method provided by the system}\}.$$

After we make a call to a method supported by a replica, the call may return one of the following:

- **OK** This means that no failure occurred during the method's execution.
- **FL** This means a failure has occurred during the execution and the method was not executed.
- **TO** This means a time-out. We use this to indicate the replica is disconnected or is down.

So,

$$c: P \times O \rightarrow \{OK, FL, TO\},$$

where $O$ is the union of all data objects managed by all replicas of the system and $c$ represents all the calls. Without loss of generality we assume that all methods are update-oriented operations. That is, if $c(p, d)$ is successful, it transforms the data object $d$ from the existing state to a new state.

We define a (parallel) transaction as $T = \{c(p_1, d_1), c(p_2, d_2), \ldots, c(p_i, d_i)\}$, where $c(p_i, d_i)$ is a call to a method (also called an operation), and $p_i \in P, d_i \in O$. For simplicity we define $d_i \neq d_j, i \neq j$ and assume that all $c(p_i, d_i)$ of $T$ can be executed.
3 ALGORITHMS FOR THE SSM

Two phases are used by the primary replica's transaction manager \( TMPR \) to process a transaction \( T = \{ c_1(p_1,d_1), c_2(p_2,d_2), \ldots, c_n(p_n,d_n) \} \) initialised from a client serviced by the primary replica. We assume that \( TMPR \) maintains a private queue for each of its non-primary replicas.

**Algorithm 1**

- **Phase 1** (execute \( T \) on the primary replica): The \( TMPR \) uses the normal 2PC protocol to execute transaction \( T \) on the primary replica.
  1. If the execution aborts, i.e., \( \exists i, 1 \leq i \leq n, c_i(p_i,d_i) = FAILOVER \), then \( TMPR \) returns an FAILOVER to the client and terminates the execution of the transaction.
  2. If the execution commits, i.e., \( \forall i, 1 \leq i \leq n, c_i(p_i,d_i) = OK \), then \( TMPR \) returns an OK to the client and the execution of the transaction continues to the second phase.

In both cases, the client continues its work after the return without waiting for the second phase execution.

- **Phase 2** (execute \( T \) on all non-primary replicas): In this phase the \( TMPR \) sends the transaction \( T \) to all non-primary replicas and forces all non-primary replicas to execute the transaction.
  1. If the execution of \( T \) on all non-primary replicas return OK, then the \( TMPR \) terminates the execution of the transaction.
  2. If the execution of \( T \) on a non-primary replica \( R_i \) returns an FAILOVER, then the \( TMPR \) periodically forces \( R_i \) to execute \( T \) until it returns an OK.
  3. If the execution of \( T \) on a non-primary replica \( R_j \) returns an TO, then \( R_j \) is regarded as disconnected and the \( TMPR \) records \( T \) and \( R_j \) into \( R_j \)'s private queue. \( T \) will be executed when \( R_j \) is re-connected to the system.

Three phases are used by the non-primary replica's transaction manager \( TMPNR \) to process a transaction \( T = \{ c_1(p_1,d_1), c_2(p_2,d_2), \ldots, c_n(p_n,d_n) \} \) initialized from a client serviced by a non-primary replica, say \( R_i, i \geq 1 \). We assume that each \( TMPNR \) maintains a private queue for the primary replica \( R_1 \).

**Algorithm 2**

- **Phase 1** (execute \( T \) on the non-primary replica \( R_i \)): The \( TMPNR \) on \( R_i \) uses the normal 2PC protocol to execute transaction \( T \).

  1. If the execution aborts, the \( TMPNR \) return FAILOVER to the client and terminates the execution of the transaction.
  2. If the execution succeeds:
     (a) The \( TMPNR \) returns a partial commit (I state to the client.
     (b) The \( TMPNR \) sends \( T \) to the primary replica's transaction manager \( TMPR \). If \( PC \) state indicates that the transaction successfully executed on the non-primary replica and is now submitted to the primary replica for propagating to all replicas. Based on the result of the primary replica execution, the \( PC \) state will be upgraded to a commit state or downgraded to an abort state (Phase 3).
     (c) If \( R_i \) is disconnected, \( T \) is stored in the private queue for \( R_i \) and will be sent to \( TMPR \) when \( R_i \) is re-connected.
     (d) The client can continue its execution after it receives a \( PC \) state. The execution of the transaction continues to the second phase.

When a non-primary replica \( R_i \) re-connects to the system, it may have missed some transactions performed by the rest of the replicas. Therefore one of the first things during the re-connection is to make sure that \( R_i \) executes all the missed transactions. This can be done through the communications between the primary replica and \( R_i \).

The algorithm 1 maintains a private queue for each disconnected mobile unit, recording the transaction \( T \) and the non-primary replicas who returned TO during the course of \( T \)'s execution. The primary replica will send all \( R_i \)'s entries, i.e., entries of the private queue for \( R_i \) to \( R_1 \) (with the same order in the queue) for execution. The \( R_1 \) will be consistent with other replicas after the execution.

Another thing that needs to be done during a re-connection is the transactions performed by \( R_i \) during the period of disconnection. In algorithm 2 these transactions are recorded in the private queue for \( R_i \). Therefore during the re-connection, if the private queue of \( R_i \) is not empty, all the private queue entries (the order is kept) will be sent to the primary replica for propagation.
4 THE MULTI-SERVICE MODEL

A more realistic assumption is that there is a set of services provided by a mobile computing system and each of these services is supported by a group of replicas. This multi-service model (MSM) can be defined by an extension to the single service model.

Formally speaking, a mobile computing system provides a set of (n) services: S = \{S1, S2, ..., Sn\}. Each service Si is supported by a set of (ki) replicas: Si = \{Ri1, Ri2, ..., Rik\}. Without loss of generality, we assume that Ri1 is the primary replica for service Si. Similar to the single service model, we also assume that primary replicas (including links among primary replicas) never fail. Let Oi = \{d1, d2, ..., di\} be the data objects managed by primary replica Ri1, then the consistency constraint requires that (eventually) for j = 2, 3, ..., ki, Oj \equiv Oi, where Oj is the data objects managed by replica Ri1.

In this multi-service model, all primary replicas are located in static units and a computing unit (static or mobile) can run a group of non-primary replicas, each of them provides a different service.

A (parallel) transaction is defined as T = \{T1, T2, ..., Tn\}, where T1 is a transaction defined in the single service model (called a sub-transaction of T). That is, all operations of T1 are addressed to a single service and can be executed in parallel. We assume that if Ti \( \notin \{T1, T2, ..., Tn\}\), then Ti and Tj address to different services and therefore all Ti's of T can be executed in parallel. The semantics of T is that, T commits if all services involved in the transaction successfully execute the operations; T aborts if any one of the services fails to execute its operations successfully.

5 ALGORITHMS FOR THE MSM

A revised two phase commit (2PC) protocol is used by the primary replica's transaction manager (MTMFP) to process a transaction T = \{T1, T2, ..., Tk\} initialised from a client serviced by primary replicas.

Algorithm 3

- Phase 1 (vote):
  1. The MTMFP sends each sub-transaction Ti \( \in \) T to the primary replica of Ti and asks the primary replica to check if the sub-transaction can be performed or not.
  2. When a primary replica receives, from the MTMFP, a sub-transaction Ti and the request to check the executability of the sub-transaction, it replies with its vote (OK or FL) to the MTMFP, based on the first phase of the algorithm TMR (Section 3). If the vote is FL, the primary replica aborts this sub-transaction immediately.

- Phase 2 (abort or commit):
  1. The MTMFP collects the votes. If there are no failures and all the votes are OK, then the MTMFP decides to commit the transactions and sends a DoCommit message to all the involved primary replicas.

- Phase 3 (propagate T): When MTMFP receives T, it uses the algorithm 3 to execute T, treating MTMFP as a client serviced by primary replicas.
6 AN EXAMPLE

We have implemented a replication manager for mobile computing based on the single service model. The system consists of two static units running on two Sun workstations and one mobile unit running on a Toshiba Satellite 420 Laptop computer.

An Oracle database running on one of the Sun workstations (Computer 1) is designated to manage data of the primary replica. An mSQL [4] database running on the second Sun workstation (Computer 2) and the Access database running on the laptop computer (Computer 3) are designated to manage data of the non-primary replicas.

A primary replica manager is executed on Computer 1 and a non-primary replica manager is executed on Computer 2 and Computer 3, respectively. The system is developed using Java language. The Java Database Connectivity (JDBC) is used to provide a uniform interface to different relational databases. Reference [8] describes some techniques we use in our implementation. We performed various successful tests on our system. These tests can be classified into the following three situations:

1. When all computers are connected and working well.
   (a) A transaction initialized from a client that is serviced by one of the non-primary replica managers.
   (b) A transaction initialized from a client that is serviced by the primary replica manager.

2. When the mobile unit is disconnected from the static units.
   (a) A transaction is initialized from a client serviced by the primary replica manager.
   (b) A transaction is initialized from a client serviced by the non-primary replica manager on the static unit.
   (c) A transaction is initialized from a client serviced by the non-primary replica manager on the mobile unit.

3. When the mobile unit is re-connected to the static units.

7 CONCLUSIONS

The difficulties of managing mobile transactions using the primary and non-primary replicas can be classified into three levels:

- Level 1: single service; the service has many replicas; a reliable primary replica; and a mobile with one non-primary replica.
- Level 2: multiple services; each service has many replicas; reliable primary replicas and reliable links among primary replicas; and a mobile with many non-primary replicas.
- Level 3: same as level 2, but primary replicas can fail and links among primary replicas can fail (network partition).

This paper has proposed solutions for the first two levels and a partial Level 1 implementation. Some further studies are carried out at this moment.

References


MODELLING AND SIMULATION OF REACTIVE SYSTEMS

CHANGGUI CHEN AND WANLEI ZHOU

School of Computing & Mathematics, Deakin University
Clayton, VIC, 3168, AUSTRALIA
Email: changgui, wanlei@deakin.edu.au

Reactive systems concepts are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms. This paper presents the modelling and simulation of reactive systems for building distributed and fault-tolerant computing applications. The Actor concept is introduced to analyse and model the reactive components. The simulation of the Actors shows the usefulness of the reactive system approach in distributed environments.

1 Introduction

Reactive systems are ones that are supposed to maintain an ongoing interaction with their environment, rather than to produce some final result on termination [6]. An adequate description of reactive systems must refer to their ongoing behaviours, which are seen as reactions to external stimuli. Typical examples of reactive systems are flight reservation systems, industrial plant controllers, operating systems, most kinds of real-time computer embedded systems, and communication systems, etc. Such systems are often concurrent and distributed [1] [7].

Several systems, such as Meta, Disco and STATEMATE, and languages, such as Reactive C and Reactive Pascal that are based on reactive system concepts have been developed recently [12]. However, most of the research on reactive systems is concentrated on process control (such as controlling a robot). In our research, we try to develop a platform based on the reactive systems concepts for building distributed and fault-tolerant computing applications. This paper attempts to model reactive systems using the Actor concept, which is a basic concept in Real-time Object-Oriented Modelling [10]. Actors are parallel autonomous agents, which are distributed in space and execute at their own rate and communicate asynchronously by sending messages. Their behaviors show that they have reactive characteristics, thus we can model reactive components as Actors. After that a simulation of these Actors written in Simjava is developed to evaluate the performance of our reactive system model. The entity concept from Simjava has similar semantics with the Actor so that Simjava provides a good simulation model for the reactive Actors.

The paper is organised as follows: In the next section, we describe the architecture of reactive systems. The Actor concept used to model the reactive components is introduced in Section 3. Section 4 addresses the reactive behaviours. The simulation of actors is presented in Section 5. Finally we conclude our work.
2 The architecture of reactive systems

A reactive system uses sensors and actuators to implement the mechanisms that interact with its environment or applications. Its decision making managers (DMMs) are used to implement the policies regarding to the control of the applications. Figure 1 depicts the architecture of the reactive system model. In this model, a DMM subscribes to sensors and receives reports from these sensors on the application's states. Then it uses actuators to change the states of the applications according to the policy it implemented. Sensors can be attached to applications to obtain their states (or events). These states or events are sent to the DMMs which react to them by using actuators to change the states of applications.

![Figure 1: The reactive system architecture](image)

With this architecture, we have a great advantage that it separates policies from mechanisms, i.e., if a policy is changed it may have no impact on related mechanisms and vice versa [12]. For example, if a decision making condition based on two sensors was "AND" and now is changed to "OR", the sensors can still be used without any changes required, i.e., the mechanism layer can remain unchanged. This advantage will lead to a better software architecture and have a great significance in developing distributed and fault-tolerant computing applications. In normal practice, most fault-tolerant computing policies are deeply embedded into application programs, therefore these applications can not cope with changes in environments, policies and mechanisms. To build better fault-tolerant computing systems that can cope with constant changes in environments and user requirements, it is essential to separate fault-tolerant computing policies from application programs. Hence, we can apply the reactive system model to develop better fault-tolerant computing applications [4][11].

3 Actor model

A reactive system has three components: DMM, Sensor and Actuator. In this section we introduce the Actor concept to model them respectively.

An Actor represents an active object that has a clearly defined purpose. It may be used to model physical objects or abstract objects. Here, the term "active" means
that an Actor may have its own execution thread and can, therefore, operate concurrently with other active objects in its domain. An Actor's behaviors illustrate that it has most of reactive characteristics. We can, therefore, model DMM, sensor and actuator as Actors respectively.

The key to identifying an Actor is its purpose. In fact, the purpose of an Actor is an abstraction, or distillation, of its various functional capabilities. This is both implied and enforced by the encapsulation shell of the Actor, which is an opaque both from without and from within the Actor.

To communicate with other entities in its environment, an Actor provides one or more interface components, which we call ports, in its encapsulation shell. The set of messages exchanged between two parties in a concurrent system typically conforms to a dynamic pattern or protocol.

We can implement three Actor classes in a reactive system: DMM actor, sensor actor and actuator actor. A DMM actor may have more than one sensor ports and actuator ports to connect to sensors and actuators, but we can define all the sensor ports using the same protocol, and so do all the actuator ports. A sensor actor has a port to connect to the DMM for reporting events and a port to connect to an application object for monitoring events. Similarly, an actuator actor has a port that connects to the DMM to receive decisions and another port that connects to an application object to change its state. The access protocols on these ports can be defined respectively.

4 Reactive behaviours

We refer to the internal operation of an Actor over time as its behaviour. At any point of time, an Actor has a state that determines how it will react when it receives a message through one of its ports. The set of such states, and the possible sequences in which the states can be visited, is described by the statechart [5], which constitutes an extensive generalisation of state-transition diagrams.

The behavioural description of the DMM actor includes following states and transitions. During its lifetime, the DMM progresses through three basic phases. On creation, it is initialised and enters "Listening" state, in which it listens to sensors connecting. Once there are sensors to connect to it (accompanying with a transition for connecting with the sensors), it enters the "Waiting" state. In this state, it waits to receive the sensor reports. If an event is received (a transition for receiving events occurs), it will enter "Operational" state, during which it makes decisions and then sends them to actuators. Hence, two transitions: "decisionMaking" and "decisionSending" happen in the "Operational" state. After sending the decisions, the DMM enters the "Waiting" state again.

The sensor actor has two states and two transitions during its lifetime, and so does the actuator actor. Their behaviours can be described using the statechart, similar to the DMM's.
5 Simulation

We develop a simulation in this section to evaluate the effectiveness of the above reactive system model. The simulation tool we use is SimJava, which is a discrete event simulation package written in Java and conceptually based on the Simlib library for C [8] [9]. The purpose of the simulation is to measure the execution time taken for communications among DMM, sensor and actuator actors. According to the time they take, we can evaluate their performance running in different environments.

5.1 Simulation model

A SimJava simulation contains a number of entities each of which runs in parallel in its own thread. These entities have the similar structure and illustrate the similar characteristics as the Actors we described above. Hence this simulation model is quite appropriate for our reactive Actors modelling.

An entity's behaviour is encoded in Java using its body() method. To simulate the Actors in a reactive system, we can create three entities to model the DMM, sensor and actuator actors. Without losing genericity, we assume that there are m DMMs and n sensors and actuators respectively in a reactive system. Each sensor can be subscribed by all DMMs and each DMM can subscribe to all sensors as well. Once a DMM makes a decision, the decision will be sent to the related actuators that change the application states.

5.2 Results and discussions

The simulation results confirm our intuition that as the numbers of DMMs and sensors/actuators increase, the execution time they used to communicate with each other increases as well.

![Figure 2: The execution times for Actors simulation](image)

Figure 2 shows the simulation results, where s stands for the number of sensors. From Figure 2, we have two findings. The first is that, the simulation time with an
increasing number of sensors increases more rapid than that with an increasing number of DMMs. In Figure 2, the values on vertical axis increase more quick than those on the corresponding horizontal axis. This is because, increasing the number of sensors, i.e., increasing applications, will increase the number of actuators so that the simulation has to take more time to receive more events and send more decisions to the actuators. This increase is by many times and is a non-linear increase, while the increase of simulation time with an increasing number of DMMs is simply by adding the times that each DMM takes together, which is a linear increase.

The second finding is that, among four time curves with different number of sensors, the bigger the number of sensors is, the steeper the curve is. That means, with the increasing number of DMMs, the simulation time with bigger number of sensors increases more rapid than that with smaller number of sensors. This can also be explained by the above reason.

5.3 Distributed simulation

To evaluate the reactive system model in a distributed environment, we have to build a distributed simulation. We can use the extension of the Simjava discrete event simulation package to utilise the Remote Method Invocation (RMI) [9] capabilities of the JDK1.2 to build the distributed simulation.

The distributed simulation shows a similar result to the local host simulation. Figure 3 shows the comparison of two groups of simulations, from which we have an observation. We can see that the time curves for the local simulation are steeper than those for the distributed simulation. That means, the increase of execution time for the distributed simulation is less than the increase for the local simulation! This is a good news for our reactive actors, because most of DMMs require reports from sensors located in different hosts. This result shows that DMMs and sensors running in a distributed environment is more effective than their running in a local host.

![Figure 3: Two groups of simulations](image)
6 Conclusion

The major advantage of the reactive system approach is the separation of policies and mechanisms. In the paper we introduced the Actor concept to model reactive systems. An Actor represents an active object and it exhibits some reactive characteristics. Hence the Actor is an appropriate tool for modelling reactive components. The simulation results show that DMMs, sensors and actuators can be used in a distributed environment effectively.

Two applications based on the reactive system concepts, one is the Teamwork support system and the other is the Network partitioning problem, can be referred to [2] [3]. They have showed the potential benefits of our reactive system approach. In both cases, the DMMs, sensors and actuators stay the same and perform the same functions no matter what changes others have. This leads to the easy design and maintenance of the applications.

References

1. P. Caspi, A. Girault and D. Pilaudi. Distributing reactive systems. The ISCA Int'l Conf. on Parallel and Distributed Computing Systems (PDCS'94), Las Vegas, USA, 1994
AN ARCHITECTURE FOR RESOLVING NETWORK PARTITIONING

Changgui Chen & Wanlei Zhou
School of Computing & Mathematics
Deakin University, Rusden campus
Clayton, VIC 3168, Australia
changgui, wanlei@deakin.edu.au
Phone: 61-3-92447373  Fax: 61-3-92447460

ABSTRACT Network partitioning is harder to deal with than many other failure problems in a distributed environment. This paper presents a solution of using reactive system architecture to solve partitioning problem with the Primary/Non-Primary replication control protocol. The reactive system concepts are an attractive paradigm in fault-tolerant computing because it separates policies from mechanisms. In the paper, DMM, sensor and actuator modules are implemented to achieve the mechanisms for failure detecting, notifying and resolving.

KEYWORDS Network partitioning, Reactive system, Fault-tolerant computing, Replication control.

1.0 INTRODUCTION

In a distributed computing environment, two types of failures may occur: the processor at a given site may fail (referred to as a site failure), and communication between two sites may fail (referred to as link failure). Link failures may result in network partitioning, which is a major threat to the reliability of distributed database systems and to the availability of replicated data [1]. Network partitioning occurs when failures fragment the network into isolated sub-networks called partitions, such that sites or processes within a given partition are able to communicate with one another but not with sites or processes in other partitions. If processes continue to operate in the disconnected partitions, they might perform incompatible operations and make the application data inconsistent [4] [5].

A number of diverse solutions have been proposed to solve network partitioning problem. These solutions can be classified as two kinds of strategies: pessimistic and optimistic [7] [8]. But most strategies on network partitioning require that the failure initially be recognized. They assume that partition failure detection has already been done. Algorithms for detecting and analyzing network partitioning, therefore, have not been developed almost. This paper attempts to propose a method which uses reactive system model to build the system architecture for resolving partitioning problem.

The reactive system concepts are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms. The development of fault-tolerant software is a very difficult task. One reason is that, in normal practice, fault-tolerant computing policies and mechanisms are deeply embedded into most application programs, therefore these application programs can not cope with changes in environments, policies and mechanisms. To build fault-tolerant computing systems that can cope with the constant changes in environment and user requirements, it is essential to separate the fault tolerant computing policies and mechanisms in application programs. The approach proposed in the paper is a relatively centralised method using the Primary/Non-Primary replication control protocol [6]. In the paper we design three reactive modules: Decision making managers (DMMs), Sensors and Actuators to implement the partition-tolerant architecture. Sensors and Actuators are used to implement the mechanisms for failure detecting and notifying. DMMs are used to implement the coordination among the replication managers.

The rest of the paper is organized as follow: in the next section we introduce the reactive system model. The architecture for partition-tolerant system is addressed in the section 3. Implementation issues will be presented in the section 4. Finally we conclude our work.

2.0 REACTIVE SYSTEM MODEL

A reactive system uses sensors and actuators to implement mechanisms that interact with the environments or applications [2] [3]. The system manages or decision making managers (DMMs) are used to implement policies regarding to the control of the applications. A DMM subscribes to sensors and therefore receives the reports of the application states. According to the reports it receives, the DMM makes decisions and uses actuators to change the states of the applications. Figure 1 depicts the architecture of our reactive system model. In this model, sensors are attached to applications to obtain their states (or equivalently, to monitor some events about the applications). These state information or events are sent to the DMMs which react to them by using the actuators to change the states of the applications. The model consists of three levels: policies, mechanisms and applications.
Policies: This level deals with policies in fault-tolerant computing. For example, it may determine what strategies are used in detecting component failures, what information is to be collected from the application programs, and what techniques are used in masking and tolerating component failures. These policies are implemented through decision making managers.

Mechanisms: This level deals with all mechanisms for implementing the fault-tolerant computing strategies. For example, it deals with mechanisms used in detecting and reporting component failures, and mechanisms used in masking and recovering from component failures.

Applications: The application level deals with application-specific issues.

![Reactive System Model Diagram](image)

Figure 1: The reactive system model

The major advantage of this model is the separation of the mechanism for gathering information and the policy for making decisions according to the information received, i.e., if a policy is changed it may have no impact on related mechanisms and vice versa. For example, if a decision making condition based on two sensors was "AND" and now is "OR", the sensors can still be used without any changes required, i.e., the mechanism layer can remain unchanged. This advantage will lead to a better software architecture and have a great significance in fault-tolerant computing [10].

3.0 PARTITION-TOLERANT ARCHITECTURE

Network partitioning most likely happens at a wide area network. We assume that the network environment is consisted of different subnets connected by gateways. At each subnet, we have database server groups which are comprised of replicas. All database servers (or replicas) store identical information initially and each of them can accept client requests (organized as transactions) that read or update stored information independently. The task of the replicated system is to maintain the data consistency among all the replicas throughout the whole network, even in the case of failures. Figure 2 shows the architecture of such a distributed replication system.

![Replication System Architecture Diagram](image)

Figure 2: A distributed replication system

Network partitioning happens when gateways between subnets fail. This leads to a situation where server group members distributed in different subnets can not communicate with one another and may stop a transaction processing. To detect partitioning failure, we embed above reactive system architecture into the replicated system. To do so, we add a dedicated decision making manager (DMM) as a server component in each subnet and it will subscribe to sensors in each server member to find out the partition existence and help in transaction processing. Sensors are embedded in each server member to report their states to DMMs.

![Partition-Tolerant System Architecture Diagram](image)

Figure 3: The partition-tolerant system architecture

For simplicity, we include two subnets connected by one gateway in our network configuration. Figure 3 shows the system modeled with the reactive system architecture. In this architecture, each server group member embeds with a sensor which reports its state to DMMs in different subnets. DMMs will decide whether a partitioning happens according to the reports received from sensors and then make decisions to instruct RPMs...
how to process transactions using actuators. In Figure 3, RPMs function for transaction processing while DMMs function for failure handling and coordination between replica groups.

Partition detecting and notifying

With the architecture in Figure 3, network partition detecting and notifying can be easily implemented. A DMM in one subnet regularly receives the reports from sensors embedded in all server members some of which may not be reachable if a partitioning occurs. If the DMM does not receive the reports from some sensors within a maximum time frame, the DMM decides that the gateway might be down by noticing those unreachable members are all located in the same subnet. To confirm the partitioning happened, the DMM sends a message to the other DMM in that subnet to see if they are reachable. If it does not receive the replied message within a maximum time from another DMM, the gateway between the two DMMs is assumed down, which leads to the two subnets being partitioned from each other.

Once the network partition has been detected, the DMM will use actuator to notify all the server groups about the partition situation to save unnecessary network communication overhead by server members trying to contact the other partitioned subnet. The DMM is also responsible to notify all parties once the crashed gateway is up and the partition no longer exists.

Partition resolving

During network partitioning, the main problem is that a client could issue a transaction request which involves server members in different partitioned subnets so that the continued transaction processing could result in data in different server members inconsistent. To solve this problem, we assume that all the Primary sites for one such transaction locate in the same subnet, which is the normal case in most transactions. Hence, network partitioning could happen in two cases. One is when a P site sends a transaction to a NP site in the other subnet. The other is when a NP site sends a transaction to a P site in the other subnet for checking and finalising it from a partial commit mode. In either case, these transactions are all sent to DMMs for recording and further processing. When a DMM receives a transaction record during a partitioning, it identifies its type, whether initialized by the P or NP site, and stores it in different object list. After the partition is repaired, different DMMs exchange their knowledge of transactions and then use actuators to instruct relevant RPMs to further proceed these transactions.

In the case where network partition results that different P sites involving in one transaction locate in different partitions, replication managers in P sites can not fully execute the whole transaction. We propose two options: one is to let the client abort the transaction and the other is to store the transaction and re-execute it after network partition is recovered.

4.0 IMPLEMENTATION ISSUES

According to the above architecture, we need to implement DMM, Sensor and Actuator classes in a replicated environment. Java virtual machines, which are rapidly becoming available on every computing platform, provide a virtual, homogeneous platform for distributed and parallel computing on a global scale, hence, we use Java language to implement them for partition detecting, notifying and resolving.

4.1 Decision making managers

DMM classes will be created and run in a dedicated host in each subnet. First, it receives the reports from sensors it subscribes to and decides whether a partition exists according to the information it receives. Meanwhile, it will communicate with other DMMs in other subnets to confirm if a partitioning really happens. If it does, the DMM will notify relevant parties of partitioning. Second, after the partition is recovered, it will exchange the transaction recording with other DMMs and use actuators to instruct RPMs to further process these transactions.

We use partition_detecting() function to achieve the partition detection procedure:

```java
partition_detecting() { while (true) {
    report_receiving(); //Receive reports from sensors.
    report_checking(); //Check if there are sensors not to report
    if (there are) {
        if (the size of a vector equals to the number of servers in that subnet, then find the DMM
            //address in that subnet.
        }
    } else {
        confirming(); //Sets above DMM as the message sending
        target and sends out checking message and waits for
        reply. If the DMM replies on time then sets the vector
        //that subnet as zero, otherwise, confirms the partition
        //and procedure.}
    }
}
```

```java
partition_notifying() { //Broadcast the partition to all server managers and DMMs
}
```

To instruct RPMs to further process transactions, a DMM sends its transaction lists to RPMs in the same subnet. For the transactions from P sites for compulsory execution, the RPMs execute them and then check the result to ensure whether it conflicts with present state. If the conflict exists, the RPMs use actuators to notify the DMM to invoke certain conflict resolving program such as a backup strategy. For the transactions from NP sites for checking and finalising, the RPMs check them to see whether they can be executed. If they can, the RPMs will check the result to see if it conflicts with the present state. If the conflict does exist, a notification should be made to the DMM and it will abort these transactions and notify the original NP sites to roll back. If no conflicts, the DMM will contact the original NP to finalise the
This paper has presented the design of the reactive system architecture, i.e., using the DMMs, sensors and actuators in each possibly partitioned subnet directed by a gateway to deal with network partitioning problem with the Primary/Non-Primary replica model. Compared with other strategies for partition problem, our method has the lower overheads and simple system architecture. Sensors/actuators can implement the failure detecting and notifying well. The DMM in each subnet is the center for failure handling and coordination among replication managers, and it separates with the sensors and actuators. This is just the advantage of the reactive system architecture.

However, one may argue that a relative centralized DMM itself is fault prone. This situation could be very rare by placing DMMs in stable sites. While the possibility does exist, transactions recorded by DMMs should be backed up in non-lost devices when they are sent by other PNP sites. Future work of system modeling and simulation is being carried out to evaluate the performance of our design of DMMs, sensors and actuators.

REFERENCES


DESIGN AND IMPLEMENTATION OF REACTIVE SYSTEMS FOR BUILDING FAULT- TOLERANT APPLICATIONS

CHANGGUI CHEN
School of Computing & Mathematics
Deakin University, Rusden campus
Clayton, VIC, 3168, Australia
changgui@deakin.edu.au
Phone: 61-3-92447373 Fax: 61-3-92447460

WANLEI ZHOU
School of Computing & Mathematics
Deakin University, Rusden campus
Clayton, VIC, 3168, Australia
wanlei@deakin.edu.au
Phone: 62-3-92447398 Fax: 61-3-92447460

ABSTRACT

The reactive system concepts are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms, which has a particular significance in fault-tolerant computing. The paper presents the design and implementation of reactive systems based on the Actor model. Three generic actor classes: Mthk, Sensors and Actuators are designed and implemented as a software package. Their application in the teamwork support system and the replicated database system has showed a good performance of our design.

1.0 INTRODUCTION

Reactive systems (see [4], [6], [8]) have been defined by Harel and Pnueli as systems that are supposed to maintain an ongoing interaction with their environment, rather than to produce some final result on termination. This distinguishes them from traditional input/output computations. Reactive systems typically contain parallelism and most often they are embedded systems. Typical examples of reactive systems are flight reservation systems, industrial plant controllers, most kinds of real-time computer embedded systems, web servers, and communication systems, etc.

A reactive system can be divided into three levels: policies, mechanisms and applications. The advantage of the reactive system method is that it separates policies from mechanisms. This advantage has a significant application in fault-tolerant computing. We know that the development of fault-tolerant software is a very difficult task. One of the reasons is that, in normal practice, fault-tolerant computing policies and mechanisms are deeply embedded into most application programs, therefore these application programs can not cope with changes in environments, policies and mechanisms [3]. To build fault-tolerant computing systems that can cope with the constant changes in environment and user requirements, it is essential to separate the fault tolerant computing policies and mechanisms in application programs.

To model reactive systems, we introduce the Actor concept, which was originally proposed by Hewitt [7], and later developed by Agha [1], to model reactive components. Actors are parallel autonomous agents, which are distributed in space and execute at their own rate and communicate asynchronously by sending messages. After modeling and designing the reactive components, we use Java language to implement the reactive system model into a Java reactive system. The Java reactive system consists of a set of object classes that forms a practical basis for developing fault-tolerant Java applications. Two case studies, the coordination mechanism for the teamwork support system even in the presence of network partitioning and the fault-tolerance in a replicated database system, have been performed to demonstrate the potential use of our Java reactive system.

The paper is organized as follow: in the next section, we address the system architecture. Modeling reactive systems is given in section 3. Section 4 presents implementation issues. The two case studies are addressed in section 5 and 6. Finally we give our conclusion.

2.0 ARCHITECTURE

A reactive system uses sensors and actuators to implement mechanisms that interact with the environments or applications. The system managements or decision making managers (DMMs) are used to implement the policies regarding to the control of the applications. A DMM subscribes to sensors and therefore receives the reports of the application states. According to the policy implemented by the DMM, it uses actuators to change the states of the applications. Figure 1 depicts the architecture of our reactive system model. In this model, sensors are attached to applications to obtain their states (or equivalently, to monitor some events about the applications). These events are sent to the decision making managers. The DMM reacts to these events by using the actuators to change the states of the applications [10].

The model consists of three levels:

- **Policies**: The policy level deals with policies regarding to the control of applications. For example, in fault-tolerant computing, it may determine what strategies are used in detecting component failures, what information is to be collected from the application programs, and what techniques are used in masking and/or tolerating component failures.
• **Mechanisms**: The mechanism level deals with all mechanisms for implementing policies such as fault tolerant computing strategies. For example, it deals with mechanisms used in detecting and reporting component failures, and mechanisms used in masking and recovering from component failures.

• **Applications**: The application level deals with application-specific issues.

![Figure 1: The reactive system architecture](image)

Communicating between actors is based exclusively on message passing. A message is a special type of data object that incorporates a mandatory message signal attribute, a message priority attribute and an optional message data object attribute. The set of messages exchanged between two parties in a concurrent system typically conforms to a dynamic pattern or protocol. This pattern defines not only which messages comprise the protocol, but also the direction and relative order in which the messages are sent and received. In essence, a messaging protocol is like a contract that constrains the behavior of both parties in a communication.

### 3.2 Actors in a reactive system

We can implement three actor classes in a reactive system: DMM actor, Sensor actor and Actuator actor.

#### 3.2.1 The DMM actor

The purpose of a DMM actor is to subscribe to sensors and receive their reports about applications’ states and then make decisions to change applications’ states using actuators. A DMM actor may have more than one sensor ports and actuator ports to connect to sensors and actuators, but we can define all the sensor ports using the same protocol called SensorListening, and all the actuator ports using the protocol called ActuatorSending:

```java
protocol class SensorListening
  in: [event, Event], [message, Message],
       [error, ErrorCode]
  out: [enable, Command], [disable, Command]
protocol class ActuatorSending
  in: [done, Message]
  out: [order, Policy]
```

where SensorListening and ActuatorSending are protocol class names, and the pairs (signal, data-type) specifies individual message data. The first parameter represents the content of appropriate data object, and the last one represents the name of the appropriate data type. Once a protocol class definition has been created, it can be used to define an actor interface which is accomplished by means of ports.

#### 3.2.2 The Sensor/Actuator actors

The purpose of a sensor actor is to monitor an application's state and report it to its subscribers. A Sensor actor has two ports to connect to the DMM and the application respectively. The access protocols on them are defined as following, where [ ... ] is the same as above:

```java
protocol class DmmReporting
  in: [enable, Command], [disable, Command]
  out: [arrival, Event], [error, ErrorCode]
protocol class ApplicationReceiving
  in: [arrival, Event], [error, ErrorCode]
  out: [none]
```

The purpose of an actuator is to change an application’s state according to the decision made by DMMs. Simi-
larly, an actuator actor has two ports to connect to the
DMM and the application respectively. The actuator
actor sends the order to the application, while receives
decisions from the DMM. The service protocols on the
two ports are similar as above.

4.0 IMPLEMENTATION OF ACTORS

Without losing genericity, we assume that there are m
DMMs and n sensors and actuators respectively in a
reactive system. The sensors can be attached to applica-
tions. Each sensor is subscribed by all DMMs so that
each DMM can receive all the application states through
classes. The applications are relevant to each other.
Once a DMM makes a decision, it will be sent to all
actuators which change the application states.

The DMM actor class

Using Java multiple threads, the DMM will create a Lis-
tener thread to build connections for each sensor. These
connections are handled by their own threads of execu-
tion. After a sensor connects to it, the DMM will create a
Receiving thread which waits for receiving application
messages from sensors using dedicated connections.
Meanwhile it will create a DmmToActuator thread that
sent decision messages to actuators after the
DMM receives a report from a sensor, and makes deci-
sions.

In order to synchronize the transmission of decisions, the
DMM creates a ThreadGroup to which each new
DmmToActuator thread created is placed into. Using the
ThreadGroup, the DMM can invoke each thread to send
the decisions to actuators at the same time.

The Sensor/Actuator actor classes

A sensor can be subscribed by many other entities. It
first makes connection requests to all subscribers with
its identification, then creates new threads to send events
cored to the subscribers. Similarly, in order to syn-
cronize the transmission of reports, each sensor creates a
ThreadGroup to which each new thread created is
placed into. Using the ThreadGroup, the sensor can
invoke each thread to report its subscribers at the same
time, rather than the threads independently having to
monitor for events and report to DMMs. In addition, the
ThreadGroup approach places the monitoring of events
in one place within the sensor, rather than each thread
having to monitor for the event which duplicates
processing.

An actuator receives decisions from DMMs, then
changes application states according to the decisions.
After that it returns a done signal to the DMMs. The
actuator class has a main actuator object that changes
application state and a waiting object that waits for
receiving decisions from DMMs.

5.0 THE TEAMWORK SUPPORT
SYSTEM

A teamwork support system in a distributed environment
can be viewed as a reactive system [2]. Normally, team
members prepare their work individually in parallel
which can be viewed as parallel steps. How to manage
such steps or sub-tasks, in another word, coordination, is
the key issue for completion of the entire task.

The most common mechanism used for coordination in
process support is a dynamic to-do list for each team
member to inform the associated sub-tasks which need
to be done. Once a team member has finished a sub-task
on the to-do list, the notification should be made to the
server. Then, at the server side, the appropriate coordina-
tion for process control can be adjusted to generate
updated to-do lists for related team members.

However, some decisions have to be made to decide how
to process next steps in the presence of network partition-
ing. For instance, when some team members who
undertake the prerequisite subtasks for a task haven't con-
tacted the server site long time, the server has to make
decisions to decide whether it should start this task
because these team members might be in a different par-
tition.

Figure 2: The architecture of teamwork coordination

The client side is the key to getting the real work done.
How to pass information between the server and clients,
I.e., download data from and/or upload data to the server
site, is very important, especially in a Web-based envi-
ronment which often has restrictions for doing so. Fur-
thermore, how to keep the system continuously running
even in the presence of network partition failure should
be considered. We embed the reactive system architec-
ture into the teamwork system and use the sensor/actua-
tor mechanism to pass information between the server
and team members for coordination, as depicted in Fig-
ure 2, where a DMM (decision making manager) will
run all the time for a teamwork-oriented task in the
server side. At the client site, sensors are attached to the
Java applet for each team member to obtain their states.

In the Figure 2, the DMM will make decisions to deter-
mine whether a sub-task can be started on the basis of
the related clients' states. The sensors will report to the
DMM the states of every client. When a policy condition
for a sub-task is satisfied, the DMM will send an order to
the related team member to start this task. Once a team member has finished a sub-task on the to-do list, the notification will be reported to the DMM through a sensor. If a partition failure occurs, the DMM will not be able to receive the reports from the sensors which are in a partition within a maximum time frame. Then, at the server side, the appropriate coordination for process control can be adjusted to generate updated to-do lists for related team members, and send to them through actuators. For the detailed coordination mechanism, please refer to [2].

6.0 FAULT TOLERANCE IN A REPLICATED DATABASE SYSTEM

Replicated systems make fault-tolerance possible [5]. Our second application is a replicated database system where two Mini SQL (mSQL) database servers (called replicas) are running on two workstations. All database servers (or replicas) store identical information initially and each of them can accept client requests that read or update stored information independently. The task of the replicated database system is to maintain the data consistency among all the replicas, even in the case of system component failures. Figure 3 shows the architecture of such a replicated system.

In this architecture, there is a replication manager running on each host where a mSQL database server is running. A client connects to a replication manager in order to obtain database services. If a client requires a read-only operation, this request will be serviced by the local replication manager by reading from the local database server. If a client wants to perform an update operation, the operation has to be performed in all database servers.

However, some decisions have to be made in case of system component failures such as a database server failure, a replication manager failure or a computer failure. Here we consider the first case, i.e., a database server failure. In this case, it is essential for a replication manager (RP) to know if a database server is alive.

For example, we assume that DB1 on Computer 1 fails. In this case, RP1 has to re-direct all requests to DB2 on Computer 2. If such a request is an update request, then RP1 has to store such an update in a stable storage (e.g., disk) and perform it on DB1 when it recovers. Similarly, when a client issues an update operation through RP2, RP2 has to store that operation in a stable storage and perform it on DB1 when it recovers.

We can use our Java reactive system modules designed previously to deal with database server failures. To do so, we run a Java sensor on Computer 1 to report the liveness of database server DB1 and another Java sensor on Computer 2 to report the liveness of DB2. Two Java DMMs are embedded in RP1 and RP2 respectively, as depicted in Figure 3. Both the DMMs subscribe to both sensors and are informed of the liveness of the two database servers and then make certain decisions to instruct RPI and RP2 to process clients’ requests (as described above).

Figure 3: Using reactive system modules

7.0 CONCLUSION

Three reactive system modules: DMM, Sensor and Actuator have been designed and implemented based on the Actor model. Their applications in the teamwork support and the fault-tolerant replicated database system have shown the potential benefits of our Java DMM, Sensor/Actuator classes. In both cases, the DMM modules stay the same. Their main task is to make decisions according to the reports from the sensors they have subscribed. This shows the advantage of separating the mechanism from the policy.

REFERENCES

A REACTIVE ARCHITECTURE FOR WEB-BASED INFORMATION SYSTEMS

Changgui Chen, Wanlei Zhou & Shui Yu
School of Computing & Mathematics
Deakin University
Clayton, Victoria 3168, Australia
Phone: 0061-3-92447166 Fax: 0061-3-92447460

ABSTRACT

Most current Web-based information systems suffer from lack of flexibility. They are hard to be applied to different environments and hard to adapt to frequently changing user requirements. This paper presents a novel approach -- reactive system approach to solve these problems. Reactive system concepts are an attractive paradigm for system design, development, and maintenance because it separates policies from mechanisms. With the reactive architecture, users can define different requirements and the Web server can dynamically process them and generate corresponding SQL codes without changing the system. The system is active and independent from specific databases, and thus can adapt to different environments.

Keywords: architecture, reactive systems, Web-based systems, database

1 Introduction

As Internet is growing tremendously, Web-based applications are becoming more and more popular [11]. A Web-based System (WS) allows users to interact with the system through a web browser, thus make it accessible from anywhere. The most well known example of WS is E-Commerce, which is expected to be the main stream shopping method in the 21st century [8] [10]. E-commerce is attractive to both merchants and consumers because of its unparalleled advantages such as accessibility, easy to use and cost efficiency.

A Web-based information system mainly involves four components: the Internet, a web browser, a web server and database(s) [3], as depicted in Figure 1. Users get on to the Internet through web browsers. After connected to the web server, the browser will allow the user to issue queries, e.g., asking for where Deakin University's year 2001 graduation commencement will be held. The browser packs the information entered by the user and sends it to the web server over the Internet. The web server unpacks the information, takes a look at what the user wants, then generates corresponding SQL command(s) and sends it(them) to the database. The database executes the query command(s) and gets the result. The result is then sent back to the web server, and reaches the browser over the Internet. Finally, the user gets the answer.

Figure 1: Web-based system model

Web-based systems are simple and convenient to use. But most of current WSs suffer from lack of flexibility [3]. They are hard to be applied to different environments and hard to adapt to frequently changing user requirements. It is very common that the structure of the WS and the database is fixed, thus the information which can be retrieved is limited. More specifically, the WSs can only provide certain fixed queries
to users and the tables in the database are usually fixed both in number and structure, hence the WS is tuned to functions on those fixed queries or tables. Access control and function authorization are usually done in a coarse grain level. Users and functions are divided into several levels, and users can only perform functions allowed for his/her level.

In this paper, we propose a novel approach, which is so-called reactive system approach, to support and increase the flexibility of the WSs. Reactive system concepts are an attractive paradigm for system design, development, and maintenance because it separates policies from mechanisms [1]. A Web-based system allows users to interact with the system through a web browser, thus it can be viewed as a reactive system. In this paper, we try to apply the reactive system concepts in the design of a Web-based information system. With the reactive system architecture, we can separate user requirements from the Web server. A pre-defined decision making manager always runs on the Web server, so that the server can process different user requests dynamically and generate SQL statements automatically to operate databases, but retaining itself unchanged. This flexible structure will let users change databases and customize their requests to make any queries through browsers. The system is also active. It can notify an administrator and end-users what happens in the system actively, rather being checked passively. Therefore, this approach can improve the system performance greatly and make the system maintenance easier.

The rest of paper is organised as follows: Section 2 briefly introduces the reactive system model. The reactive architecture of the Web-based system is addressed in Section 3. Section 4 discusses the dynamic generation mechanism for database operations. The implementation issues is presented in Section 5. Finally, we summarise our work.

2 Reactive System Model

Reactive systems have been defined by Harel and Pnueli [6] as systems that are supposed to maintain an ongoing interaction with their environment, rather than to produce some final result on termination. Such systems are often concurrent and distributed.

Reactive systems are everywhere. Typical examples of reactive systems are flight reservation systems, industrial plant controllers, operating systems, most kinds of real-time computer embedded systems, web servers, and communication systems, etc. Common to all of these systems is the notion of the system responding or reacting to external stimuli, by sending signals and commands to its environment.

The reactive system approach views a system as a reactor that continuously interacts with its environment by receiving and sending messages, which is the nature of a reactive system. According to different layers of software processes, a reactive system can be divided into three levels: policy, mechanism and application. Usually, a reactive system uses sensors and actuators to implement the mechanisms that interact with its environment or applications; its system controls or decision making managers (DMMs) are used to implement the policies regarding to the control of the applications [1]. Figure 2 depicts the architecture of the reactive system model. It has three layers: policies, mechanisms and applications. These layers interact with each other in the system. The policy layer, which is for making decisions, is based on the mechanism layer to sense and actuate application objects.

In Figure 2, a DMM subscribes to sensors and receives the reports of the application states. According to the policy implemented by the DMM, it uses actuators to change the states of the applications. Sensors or actuators are software entities to implement all inputs or outputs between the DMMs and the application objects.
Application objects can be any objects related to the environment. In this model, sensors are attached to applications to obtain their states (or to monitor some events about the applications). These events are sent to the DMMs. The DMMs react to these events by using the actuators to change the states of the applications.

![Diagram of the reactive system architecture]

**Figure 2: The reactive system architecture**

The above reactive system architecture has a great advantage that it separates policies from mechanisms, i.e., if a policy is changed it may have no impact on related mechanisms and vice versa. For example, if a decision making condition based on two sensors was "AND" and now is "OR", the sensors can still be used without any alterations, i.e. the mechanism layer can remain unchanged. This advantage will lead to a better software architecture [12].

3. System Architecture

The current Web-based system model is the Browser/Server model [5], which has three layers: a browser, a middleware and a database server. The Browser at the client site is the thinnest layer and it is the interface to users. The middleware runs on the Web server, and it is a program providing a bridge for user to access the database server. The database server is normally a commercial database management system, which deposits and manages the data.

In a Web-based environment, there are so many users, who may be dispersed in all round the world and have different requirements, accessing the Web-based information systems [2]. At the present, most WSs are developed according to the limited user requirement analysis. The database applications have to be changed even when there is a little difference on the requirement analysis. Furthermore, the system administrators need to modify or add new programs if there are some changes on database objects, such as adding a new table, or adding a new attribute to an existing table, etc. [4] [7]. Therefore, it is very desirable that we can develop a flexible database application that can cope with any changes on the users' requirements or database objects while remaining itself unchanged.

To achieve this goal, it is necessary to provide a method by which users can express their own requests and can customise the results they want. One way is that we present all the database information to users first and then provide an interface to them for defining the requirements. After finalising their requests the users submit to the Web server. Therefore, the Web server can process the requests and translate into SQL statements dynamically and then submit to the database server. The key problem here is that how can the Web server translate the different users' requests into SQL commands dynamically and correctly?

To solve this problem, we embed the reactive system architecture into the Web-based system and use a DMM to implement the dynamic SQL generation mechanism and sensor/actuator mechanism to implement the communication between the Web server, users, and DBMS, as depicted in Figure 3. Figure 3 shows the system structure modelled with the reactive system architecture, where at the Web server site, a DMM runs all the time and is dedicated to the processing of users' requirements and translating them into SQL statements. At client sites, sensors are attached to browsers as Java applets downloaded from the Web server for each user to obtain their information and then send to the DMM. The DMM will react to them by generating
SQL commands dynamically and then use the actuator to submit to the DBMS. A sensor is also attached to the database to monitor some events so that the DMM can notify users actively when such events happen in the database.

In the architecture, the DMM is the core of the system. The DMM has following functions: (1) it provides table information from the database to users, so that users can know what they can do; (2) it generates dynamic SQL commands according to the user requirements, and then submits them to the database server for execution; (3) it keeps track of user states and notifies them when some events occur.

![Figure 3: The system architecture (A – Actuator, S – Sensor)](image)

The flexibility is gained from this architecture, in which users can define different requirements and the Web server can dynamically process them and generate SQL commands. In contrast to other applications, which have the fixed SQL codes and can only process certain fixed queries, our system can meet different user requirements and adapt to changing environments flexibly. The DMM at the Web server site makes the whole system reactive. The separation of sensors and DMMs enables the system to be more flexible and easy to maintain. In the system, the DMM stays same no matter what sensors are. One can change sensors or applets while retain the DMM at the server site unchanged, and vice versa. This is the benefit of our reactive system architecture.

The system is also active. A good system should let users know when an important event happens instead of waiting for users' constant checking of system status passively. It would be desirable if the system can notify users when it is necessary. For instance, an email can be sent to him/her if the quantity of a certain product falls to very low level and may need to be reordered. In our system, the event sensor attached to the database implements such function. These events can be set as some conditions, such as "the number of computers becomes less than 10", etc. Once such events occur, the sensor will report to the DMM and then an email will be sent out to the users. We can even let users set these event conditions in an interface.

4 The Generation Scenario

The major advantage of our system is that the system can be dynamically reconfigured, i.e., the DMM at the Web server can generate SQL commands dynamically according to users' requirements. We divide user requirements into two categories: database changes and information processes. Database changes include adding or deleting tables, or attributes of the tables. Information processes include users' queries and insert, delete and update tables' contents. For all the changes, the system can dynamically generate corresponding SQL codes, rather than changing the system and writing SQL commands manually. Our approach is divided into two steps as follows.
First, we design two system-level tables for the DMM to catch the database information from the DBMS: one is *Tb_relations*, which describes the tables in the database and the relations between the tables; another is *Tb_attributes*, which presents the attributes of the tables, as depicted in follows:

- *Tb_relations* (TableName, PrimaryKey, ForeignKey, ForeignTable);
- *Tb_attributes* (TableName, AttrName, DisplayLabel, DataType, Length);

where TableName is the name of the tables in the DBMS and ForeignTable is the name of the relevant foreign tables; AttrName is the name of the table attributes; DisplayLabel is the label to be displayed for attributes; DataType is the data types of attributes; Length is the lengths of attributes.

The DMM extracts the database schema from the data dictionary of the DBMS, and it makes the system independent from a specific DBMS. In fact, we can get the above database information from the dictionary directly, but it is dependant on the specific DBMS [9]. This prevents the system to be used on other DBMSs. In our system, the only thing that a DB administrator needs to do is to maintain the two tables of the DMM. Hence, our system is more generic and totally independent from specific DBMSs.

Secondly, the DMM will produce SQL commands dynamically based on the structure of the target tables and user inputs. The DMM will typically implement the following:

- Provides user table information from *Tb_relations* and gets user demands for database maintenance or information queries.
- Checks *Tb_attributes* and user demands to decide what user input fields should be provided in the interface.
- Gets user input.
- Checks *Tb_attributes* and user input, decides what tables and attributes the operation is on.
- Constructs user requirements.
- Generates corresponding SQL commands.
- Sends SQL commands to the database server for execution.
- Returns the result back to the user.

Currently, most relational DBMSs have the optimisers, which can optimise SQL statements, therefore the DMM does not need to do any query optimisation for the SQL statements it dynamically generated.

We will discuss the detailed implementation of the DMM, sensors and actuators used in the Web-based database system next.

5 Implementation Issues

We use Java to implement our system in this section. The Java programming language, which has the capabilities of delivering applets over the Web as well as the claim of writing the code once and then running anywhere [12], has encouraged us to develop Web-based systems in Java. Besides a Java server application and a Java applet for users, we implement a Java decision making manager class at the Web server to coordinate user requirements and a Java sensor and actuator class respectively to pass information between the server and users.

5.1 DMM Class

A DMM class is embedded in the Web server. Using Java multi-threads feature, the DMM class is created as a multi-threaded entity. When the server runs, the DMM object will be created. Then it subscribes to sensors and waits for their connection requests. Once a sensor is connected to it, the DMM will create a thread to manage the communication between this sensor and the DMM, thus the
DMM can receive messages from all sensors using dedicated connections established between it and sensors. Each connection is handled by its own thread of execution.

In practice, when a user accesses to the Web server s/he can issue his/her requests involving the database operation on a Java applet downloaded from the server site. The DMM will build a connection with this applet and create a thread to manage the communication between the user and the Web server. A sensor is built on the applet and caches all the information the user inputs and reports to the DMM. The DMM will record this user's status and present all the tables and attributes information from the two system-level tables to the user, so that s/he can know what s/he can do on the database. The DMM also provides an interface for the user, with which the user can define the select list and the filter conditions. After the user defines his/her requirements and sends to the DMM, the DMM constructs SQL commands automatically based on these requests, then submits them to the DBMS using the actuator. The DBMS then executes the SQL commands and returns the results to the user. If the user exits the system, the thread created for him/her in the DMM will be closed and the resources are released.

5.1.1 Database Changes

When a high-level user wants to add or delete a table, s/he will be prompted an interface which provides all table items from Tb_relations. We may call this interface as an item manager. The user can choose "Add" function or simply choose a target table ("Delete") from the item manager. If s/he chooses to add a new table, then another window is prompted for attribute design. Also "Edit" function for an existing table fields is provided in the item manager for attribute changes. After the user finishes attribute design, s/he returns to the item manager. From there, a click on "Build" button will cause the DMM to generate the create-table SQL command and send it to the database server to build the actual tables and related triggers on them.

5.1.2 Information Processes

First, the user is prompted to choose target tables from an item list. The DMM then checks Tb_relations to get the table names the user is querying. It then checks Tb_attributes to get all or part of attributes of these tables, which the user is interested in. With such information, the DMM provides an interface to allow the user to input the search criterion. When the user clicks "Get" button, the DMM builds and issues a SELECT command of the following format:

```
SELECT field1, field2, ...

//all or part of attributes
FROM table1, table2, ...

//table names from the item list
WHERE
  field1 = "criteria1" AND/OR
  //user inputs criteria, and selects
  field2 = "criteria2" ...

//relevant logic operators.
```

The system also provides INSERT, UPDATE, and DELETE commands.

5.2 Sensors and Actuators

Sensors and actuators used in the Web-based system are simple. Firstly, after a client applet connects to the server, a sensor object will be created at the client site attached to the applet. Each applet will use the sensor to catch user's information and report it to the subscriber (DMM) using the dedicated connection established between the sensor and the DMM. Sensors used in the Web-based system are a kind of event sensors, which means that sensor objects run and listen to the applet's events all the time and report to the DMM immediately once any event occurs [12].

The sensor class includes a main sensor object that monitors for events, a listener object that listens for connection requests from DMMs, and a collection of zero or more member objects, each representing a
connection to a DMM and responsible for communication between the sensor and the DMM.

An actuator object will be created at each database to forward SQL commands received from the DMM. The actuator class includes a main actuator object that sends the commands to the DBMS and a receiving object that receives the commands from the DMM. Readers who are interested in the details of sensors and actuators can be referred to [1].

6 Conclusion and Future Work

The paper has presented an architecture based on the reactive system model to increase the flexibility of Web-based information systems. With the reactive architecture, the system can cope with frequently changing user requirements and adapt to different environments. Users can define different requirements and the system can dynamically process them and generate SQL statements. The system is also active. It can notify users actively about its changes. The separation of sensors and DMMs enables the system to be more flexible and easy for maintenance once there is a need to change. In the system the DMM can retain the same no matter what changes the sensors have, and vice versa. This is the benefit of our reactive system approach.

Future work for the optimisation and evaluation of the system will be carried out soon.

References


1997 3rd International Conference on

Algorithms And Architectures
for Parallel Processing

ICA3PP

Melbourne, Australia
December 10–12 1997

Editors
Andrzej Gosinski, Michael Hobbs, Wanlei Zhou
Deakin University Victoria, Australia

Sponsors
Deakin University
Faculty of Science and Technology, Deakin University
IEEE Victorian Section

World Scientific
Singapore • New Jersey • London • Hong Kong
Chapter 1
Basic Issues of Algorithms and Architectures for Parallel Processing

Andrzej Goscinski, Michael Hobbs, Wanlei Zhou

Deakin University, Australia

Demand for high-performance computing has been growing recently at an enormous rate. These days parallel computer systems are not only used in academic and government research laboratories, but also in industry and commercial environments. High-performance computing depends on parallel computer architectures, their operating systems, software support for parallel programming, and demanding scientific and engineering applications.

1 Practical Parallel Architectures

Parallel computing has many different forms which are achieved by building a variety of computer systems following some models of parallelism. Different types of parallelism are defined using a model based on two different streams used in the computation process. These streams are: a sequence of objects such as data, and a sequence of actions such as instructions. There are four combinations of these streams which lead to four models of parallelism and basic computer system architectures:

- SISD - single instruction, single data. This model does not have obvious parallelism. However, parallelism, called instruction-level parallelism (ILP), could be achieved at a very low instructional level through multiple actions being taken on objects.
- SIMD - single instruction, multiple data. This model and the computer system architectures upon which they are built address directly basic data structures such as vectors and matrices. There are two architectures which are implementations of this model: array processors (massively parallel processors) and vector processors.
- MISD - multiple instruction, single data. There has been little interest in implementing this model, since there is no programming construct to map programs into computer architecture.
- MIMD - multiple instruction, multiple data. There are a variety of architectures of MIMD based systems ranging from traditional tightly coupled processors to loosely coupled workstations connected by a local area network. These architectures differ mainly in the interconnection network between processors.

There are the following four practical system platforms: Symmetric Multiprocessors (SMPs), Massively Parallel Processors (MPPs), Distributed Shared Memory Systems (DSM), and Clusters of Workstations (COWs).

Symmetric Multiprocessors - In SMP systems all processors are connected to a
globally shared memory through a high-speed bus. These processors are usually vector processors. Each processor has equal access to the shared memory, I/O devices and operating system, resulting in a symmetric architecture. This allows a medium grained level of parallelism to be achieved. This architecture does not scale well because it uses a centralised shared memory. Examples of SMP systems include: Cray CS6400, IBM R30, and DEC Alphaserver 8000.

Massively Parallel Processors - MPPs are build using commodity microprocessors as processing nodes with memory physically distributed over these diskless nodes. These memory “modules” can be shared if an application requires it. The computers are connected to a proprietary high speed network using a tightly coupled interface, connected to the memory bus of a node. Communication in MPP systems is based on message passing or shared memory. The MPP system scales well and supports medium and fine grained parallelism. Examples of MPP Systems include: Intel TFLOP and Paragon, Cray T3D/T3E, and IBM SP2.

Distributed Shared Memory Systems - In DSM systems each processing node has a memory and cache module. This implies that memory is physically distributed among the different nodes and is shared. An illusion of a single address space for application programmers is created by the system hardware and software. Since connections between processors are not fast enough, these systems favour algorithms that rarely refer to memory locations that processors do not own. All processing nodes are connected to a proprietary high speed network using a tightly coupled interface. Examples of DSM Systems include: Stanford DASH; and Cray T3D and T3E.

Clusters of Workstations - A COW is a set of workstations connected through a low-cost commodity network, such as Ethernet, FDDI or ATM. Memory is distributed and is not physically shared. The peripheral devices are usually not part of individual workstations; they are individually connected to a network and shared, although workstations can have private disks. The network interface is loosely coupled to the I/O bus. In a COW a copy of the operating system (network or distributed) resides on each workstation and communication in usually based on message passing. The COW model scales well and supports coarse grained parallelism. Examples of COWs include: the MOSIX System, the Berkeley NOW, and the RHODOS System.

However all the excellent architectural and computational features of parallel computer systems suffer from the lack of good parallel software. As the available parallelism of computer systems increase, the problem that remains is how to exploit this parallelism. One aspect of this problem is first how to find and express parallelism, and then how to manage it transparently.
2 Operating System Support for Parallel Processing

Exploiting the increased parallelism offered by parallel computer systems, in particular distributed shared memory multicomputers and COWs, becomes the limiting factor. We have identified two basic factors that limit the writing of efficient application software:

- finding and expressing large degrees of parallelism, which is an algorithmic and/or partitioning problem; and
- managing efficiently the available parallelism in order to achieve high performance, which is a scheduling or placement problem.

In order to manage efficiently the available parallelism to achieve high performance, to save programmer's time and effort, and to avoid common errors (that are notoriously hard to find) in code with explicit communication primitives:

- a parallel execution environment should be provided by a distributed/parallel operating system \(^{10,11}\) which is responsible for transparent mapping of processes to, creation of processes on, and moving processes between workstation processors (machines); communication between concurrently executing processes; group communication; computation coordination; and distributed shared memory;
- these operations should be performed based on information generated by a parallel programming language, a parallel programming tool, or a compiler.

The objective being to provide these services in such a manner that both high performance and parallel execution transparency are achieved.

2.1 Parallel Execution Transparency

By the operating system providing transparent interprocess communication we are able to uniformly address the coordination and resources access issues required to provide parallel execution transparency, including:

- Location transparency: the whole multicomputer or COW looks like a single powerful machine rather than as a set of connected machines. This implies that the programmer does not need to know the location of an object within the parallel computer system they wish to communicated with;
- Process relation transparency: parent/child relationships are maintained over local and remote machines, ensuring that the return values of child processes are collected and presented correctly to the parent process. This implies that the programmer does not need to be involved in the computation coordination; and
- Device transparency: access to devices such as the screen, terminal, I/O ports and files is via message passing and thus, is also location independent. This implies that the programmer can control the parallel computation from their home machine.
2.2 Services of an Operating System Supporting Parallel Execution

In order to have both transparent parallel execution of user programs on a parallel computer system and efficient load balancing to ensure an even system wide load to achieve high performance parallel execution, global scheduling should be transparently employed to allocate processes to machines. Global scheduling should adaptively utilise both:

* static allocation, supported by remote creation of processes when computation load is steady, to perform initial allocation of processes to machines; and
* load balancing, supported by process migration for when the computational load continually fluctuates and must be made even. This includes the case where locally generated processes are moved to a remote machine to balance the load induced by parallel execution.

The allocation of processes to machines must be performed in the fastest possible way in order to achieve high performance execution. However, we believe that an operating system should be constructed to provide such services as remote process creation and process migration transparently. The performance of these mechanisms are crucial to the successful implementation of a parallel execution environment.

A fundamental parallel programming construct is that of a sequential process creating a set of concurrently executing sequential processes to achieve some objective, referred to as nested parallelism. This construct can be implemented either by creating a set of processes locally, and then migrating them to selected machines; or remotely creating a requested set of processes on selected machines. These operations can be performed using either a one-to-one communication pattern or group communication. The latter dramatically reduces the communication overhead.

The provision of transparent resource sharing in a DSM multiprocessor or COW could, if not addressed properly, cause both coordination problems when program units execute in parallel, and the loss of control by users over their resources. Computation coordination could be provided by exploiting the definition of a "home" machine and communication transparency. According to this concept, all the user's processes seem to run at the user's "home" machine forming a parent/child relationship. This implies the programmer does not need to know process locations to support their coordination.

In order to allow processes executing on different machines to share memory, a DSM system provides the abstraction of a globally shared memory. Programmers are unaware that the underlying mechanism for interprocess communication is based on message passing and are able to write their code using well known shared memory primitives.
3 Support for Parallel Programming

Support for programming parallel applications on parallel computer systems must address two major issues generated by distribution: the ability to execute units of a program on different workstations in parallel, and the ability for these units to communicate and synchronise. There is also a third issue, the ability to deal with partial failures of parallel computer systems. However for parallel applications, fault tolerance, which is very expensive to support, is of minor importance and should not be provided at the cost of a significant decrease in performance. Programming support for parallel processing should be addressed by using one of the following four approaches:

- Parallel/Distributed programming languages, which above and beyond other programming issues must address parallelism, communication and synchronization of units of parallelism of a program. These languages provide a parallel execution abstraction which allows users to explicitly specify parallelism. However by definition they require a programmer to identify units of parallelism;
- Parallel programming tools, which exploit sequential languages (e.g., C) and include some library routines that invoke operating system facilities. These primitives allow a programmer to specify units of parallelism in a program and support communication and synchronization between these units. These tools require a programmer to identify units of their sequential programs to be executed in parallel;
- Parallelizing compilers, i.e., compilers capable of identifying both units of parallelism in a sequential program written in procedural languages (e.g., Fortran, C), and the communication/synchronization points in these units; and
- Sequential programming languages, supported by a virtual global shared memory (DSM).

Parallel processing of programs can be classified by the grain of parallelism. The grain can be assessed by using the amount of computation time between communication operations of units of parallelism (e.g., process) of a program. Coarse grained units of parallelism are computation oriented, spending most of their time doing computations and rarely communicate. On the other hand, fine grained units of parallelism communicate very frequently.

3.1 Parallel (Distributed) Programming Languages

Program partitioning and restructuring for parallel execution can be carried out using distributed and parallel programming languages. During the previous decade a huge number of languages for parallel computers and COWs (distributed systems) have been implemented. Several models have been developed for expressing parallelism and for supporting communication and synchronization, and are used in these
languages. The granularity of parallelism of these languages varies.

At this stage of development of languages for parallel/distributed programming, processes and objects are the units of parallelism which are used. Mapping units of parallelism onto workstations is closely related to expressing parallelism and is a natural consequence of the distribution of computational resources (processors or workstations). There are two basic approaches to mapping: some parallel languages allow programmers to allocate units of parallelism onto processors/workstations; whereas other languages relieve programmers from this task, instead employing a compiler and run-time system to perform the mapping.

Units of parallelism can be assigned at compile time, run time or dynamically. There is one advantage of the first approach, namely that if there are some units of parallelism which can run on the same machine, a programmer can take advantage of sharing memory between these units. However each parallel program is treated by programmers independently, and as such programmer and compilation controlled activity could generate a huge computational load imbalance. Furthermore, it should be the task of an operating systems, not compilers or programmers, to map units of parallelism to workstations.

Following the run-time approach, units of parallelism are allocated to machines upon creation. This requires an operating system to support remote process creation and high level (global) scheduling. There are only a few parallel languages which support the third approach and allow units of parallelism to execute on different machines during their lifetime. This approach is very much oriented toward high performance computation on CONs.

Communication and synchronization in parallel languages is provided by message passing or data sharing. Message passing can be supported by: synchronous point-to-point primitives; asynchronous point-to-point primitives; rendezvous mechanisms; remote procedure calls (RPC); or one-to-many (group communication). Data sharing may exploit distributed data structures or logical variables.

Parallel languages are divided into two groups: distributed address space languages (fully parallel languages) whose modules do not share an address space and communicate only by message passing; and shared address space languages, with a shared (global) address space. The second group include languages such as functional languages, logic languages and distributed data structure languages.

In summary there are some evident advantages to be found in parallel programming languages:
- a language may provide a programming model of a higher abstraction than the message passing model, used by operating system supporting parallel programming tools;
- data structures and data types are adequate for parallel programming, which is not always a case of the approach based on parallel programming tools;
an ability to deal easily with data types; and
a language offers good readability and portability.

However, many existing parallel programming languages are somewhat difficult to use — an application programmer must learn another language when they have little prior experience with parallel programming, and dealing with many languages is not a major part of their activity. These facts limit the acceptance of parallel programming languages among computational scientists. Application programmers have moved toward parallel programming tools based on sequential languages supported by an operating system.

3.2 Parallel Programming Tools

The application programmer is able to perform the task of parallelizing a program. This typically requires identifying the units of the program which can be executed in parallel, and providing the management and/or coordination code that must be performed in order to create processes, send and receive messages, coordinate the execution of parallel units, and to collect results. These operations supporting a programmer require some special programming tools.

A number of such tools have been developed and successfully used, in particular PVM\textsuperscript{14}, MPI\textsuperscript{15} and p4\textsuperscript{16}. Packages such as PVM, MPI, and p4 require a programmer to identify units of their programs to be executed in parallel, and apply some parallel execution primitives to support coordination and communication between these units. These packages are very much communication oriented, with various levels of support. However, a programmer must deal with the details of data, in particular their types, which complicates building parallel programs.

Process management is provided differently by these three tools. PVM supports local and remote process creation and termination. p4 provides limited support for remote process creation — a process created dynamically is able to only interact with its parent process with whom it shares memory. On the other hand, the MPI standard does not provide remote process creation and also does not define any underlying mechanisms — only an interface is defined.

With these tools a programmer can modify the overall structure of a program, or substitute existing algorithms with equivalent algorithms, in order to make the program perform better in a parallel computing environment.

In summary, the major disadvantage of these tools is that programmers are forced to identify units of parallelism in a sequential language and then to specify how parallel machines executing these units cooperate over the execution course. Additionally, the mapping of processes to machines must be carried out by the programmer and any necessary communication must be managed explicitly. Such hand-coding is time-consuming and error-prone. In most cases parallel codes generated in such a way are rarely portable across different systems. A challenging approach to
relieving programmers from these tasks is automated parallelization of sequential code.

3.3 Parallelizing Compilers

The task of parallelization can be split into three sub-problems\(^\text{17}\): identifying the potential units of parallelism (either automatically or by a programmer); mapping these units on to the target computer system; and generating and optimizing the parallel code. Even when all the parallelism is identified by a programmer, the mapping and code-generation requires sophisticated compiler analysis and program transformation.

Automatic detection of parallelism has been studied extensively over the last two decades. Current research efforts in automatic program parallelization have only concentrated on exploiting loop-level parallelism\(^\text{18}\), surveyed in Bacon et al.\(^\text{19}\). This has been implemented on several existing systems, and can be shown to work for a range of problems with a mathematical basis\(^\text{20}\). Significant progress has been achieved in recent years with fine-grained parallelism\(^\text{21}\).

As a result of this work, current parallelizing compilers are mostly oriented toward optimizing control flows over regular data structures on parallel computer systems with a shared global address space. They have limitations when parallelizing loops with indirect access patterns. Automatic parallelization of sequential programs on distributed memory computer systems is far beyond the capability of today’s compilers, mainly because of their weakness in orchestrating data\(^\text{22}\).

Existing compilers either map processes to machines based on information provided by a programmer, or they force the programmer to map processes and data to machines, which is a major disadvantage of this approach. Another problem is that compilers do not force operating systems to take responsibility for communication and synchronization between processes. These operations should be performed based on information generated by a compiler. However, as yet there is no compiler capable of taking maximum advantage of parallelism by generating information which can be used by an operating system to support parallel execution.

The main disadvantage of parallelizing compilers is the identification of program units which can be run in parallel. This is associated with the granularity of parallel execution, in particular for COWs that can efficiently support coarse-grained parallelism. Coarse-grained parallelism can arise when two or more statements of a program (i.e., basic blocks, different loops or functions) are executed in parallel.

Coarse-grained parallelism, unlike regular loop parallelism, is more difficult to detect, package, schedule, or even express in a high-level language\(^\text{18}\). Exploiting coarse-grained parallelism requires synchronization between control and data dependent tasks, and should be based on basic language and compilation constructs.

However, research into parallelization must address not only how to identify
both coarse grained program units which can run in parallel, but also how to identify synchronization and communication points in programs. How to automatically extract information about these units and communication requirements for a global scheduler of an operating system also must be addressed.

Efforts in building parallelizing compilers has concentrated on one language, Fortran, since it has been extensively used for huge computations, and one parallel system platform, SMP, because it provides physically shared memory. However some projects have recently been initiated to address DSM systems\textsuperscript{23} and COWs\textsuperscript{12,13}.

Parallelizing compilers for DSM systems and COWs are very promising tools because they could relieve programmers from the time consuming and error prone activities of manual parallelization and coding with explicit communication operations. By automating the parallelization process, the scientific computing community can be offered high levels of performance. However, parallelizing compilers for DSM systems and COWs are at the beginning of the development path, and require more theoretical and experimental research before arriving at an acceptable product.

3.4 Sequential Programming and DSM

Programmers know how to write sequential programs to solve a variety of problems, including large scale scientific computation problems, and successfully practice this activity. A lot of high quality software has been developed over the years. The assumption used in developing this software was that there is a physically shared memory. Programmers would be very happy if they could write new software and execute existing software on parallel computer systems such as distributed shared memory systems and COWs using the shared memory approach. However these two classes of parallel systems do not provide physically shared memory.

In order to allow programmers to use existing application software and to write new application software in the form of sequential programs a proposal has been made to provide a logical distributed shared memory (DSM)\textsuperscript{23}. DSM allows processes to assume a globally shared virtual memory even though they execute on machines that do not physically share memory. Thus, programmers can focus on solving the problem, and do not have to be concerned with managing partitioned data sets and communication values.

Hence this approach to support programming for parallel systems does not require anything different to that of sequential programming. A DSM system should be developed in such a way that programmers are not forced to use any new (special) constructs to write and execute their programs in a parallel environment. For this reason a DSM system should be integrated into an operating system.
4 Parallel Applications

Traditionally parallel computer systems are used to solve computationally intensive problems in areas such as computational fluid dynamics, climatic modelling, molecular biology, chemical structure modelling, and so on. Nowadays these systems can also be used for solving large number of problems that are not numerical in nature.

4.1 Parallel Algorithms

The ability to find solutions to the variety of numerical and non-numerical problems, for which parallel computers are increasingly being used, depends greatly on the employed parallel algorithm. Algorithms developed for parallel computers, either supported by a compiler or developed by hand, are difficult to produce and often complex in structure. Therefore parallel algorithms have been the focus of considerable research recently with the objective to simplify the procedure of developing parallel algorithms and to increase the range of problems for which parallel algorithms can be created.

Currently, a variety of methods exist for the development of a parallel algorithm for a particular problem, including: designing the algorithm with a specific parallel computer system or architecture in mind; converting a sequential algorithm into a parallel version; and, writing the algorithm from scratch. Ideally, the programmer should not be concerned with the final parallel architecture on which the algorithm will execute; this should be transparent to the programmer. Nor should the programmer have to totally re-write a program just to run on a parallel computer. Therefore, to simplify the development of parallel algorithms the programmer requires support at a variety of levels, including the compiler, language and development levels.

The complexity of a parallel algorithms can be analysed in two dimensions. The first dimension is the total time required to find the solution, and the second one addresses the space (amount of temporary storage) required to find the solution.

4.2 Support for Parallel Applications

Parallel computer systems that are used to support today's applications can be categorised as general purpose parallel computers or special purpose parallel computers. General purpose parallel computers can be efficiently used in a wide range of applications, including artificial intelligence, scientific visualization, finite element analysis, optimization, commercial transaction processing, simulation, and signal processing, to name a few.

Special purpose parallel computers can be used to further improve the performance of applications. However these specially developed computers are usually expensive because they can only be used efficiently in the special areas they are designed for, while general purpose parallel computers can be used by many different applications.
A bridge between the previous two approaches are the reconfigurable massively parallel computers, in which processors are allowed to select different local connectivity to accomplish a desired global topology. Such reconfigurability naturally facilitates the mapping of a parallel application to the architecture best suited to solving the problem.

4.3 Support for Distributed and Parallel Databases.

Parallel computer systems and clusters of powerful workstations have a significant impact on distributed and parallel database systems. COWs provide the architecture for distributing database functions among workstations, where data access functions are handled by dedicated powerful computers, called data servers, while the user interface and presentation functions can be executed on any workstation. Furthermore, some workstations can also be used to perform general and specific computational and application functions, such as providing transparent, reliable, secure, and efficient distributed services. This architecture is constantly referred to as the three-tier architecture.

To improve performance and data availability, a data server can be implemented on a general parallel computer system or a database machine. Running a data server on a general purpose parallel computer system has the advantage of adopting existing parallel processing techniques for parallel data management. Database systems, especially relational database systems, provide several opportunities for parallelism, notably: inter-query parallelism (parallel execution of multiple queries), intra-query parallelism (parallel execution of multiple operations of a single query), and inter-operation parallelism (parallel execution of multiple sub-operations of the same operation).

A database machine is a dedicated system capable of performing all or part of database management functions in hardware, software, or a combination of both. The main advantage of database machines is the potential of overcoming the inefficiency of using conventional machine to handle database operations. As a result, database machines are parallel systems capable of performing database operations and data access functions efficiently.

Another way of improving performance of data servers is to associate a general purpose parallel computer with some special purpose devices dedicated to perform some database functions.

5 References


2000 4th International Conference on
Algorithms and Architectures
for Parallel Processing

ICA³PP
2000

Hong Kong
December 11-13 2000

Editors
Andrzej Gosciniski
Deakin University, Australia

Horace Ip
City University of Hong Kong, Hong Kong

Weijia Jia
City University of Hong Kong, Hong Kong

Wanlei Zhou
Deakin University, Australia

Sponsors
Deakin University, Australia
City University of Hong Kong, Hong Kong
The Croucher Foundation
IEEE Hong Kong Section

World Scientific
Singapore • New Jersey • London • Hong Kong
Preface

Welcome to The fourth IEEE International Conference on Algorithms and Architecture for Parallel Processing (ICA3PP2000).

Cluster and scalable parallel computing have proved their abilities in attaining very high performance at very low cost for information processing. With the rapid enhancement of the bandwidth of interconnection networks and performance of PCs and workstations, as well as the improvement of operating systems and software, parallel processing and distributed computing are becoming more cost-effective. The objective of ICA3PP2000 is to bring together researchers all over the world to advance the theories and technologies in parallel and distributed computing. The main focus of ICA3PP2000 is on two broad areas of parallel and distributed computing, i.e., architectures, algorithms and networks, and systems and applications.

The conference committee received 127 full manuscripts from researchers and practitioners of 15 countries. Each paper is sent to at least three international reviewers. Papers were reviewed and selected based on their originality, significance, correctness, relevance, and clarity of presentation. From the high quality submissions, 34 long papers and 26 short papers were accepted for presentation at the conference. That represents the acceptance rates of 26.8% and 47.2% for long and short papers, respectively. All the accepted papers are included in the proceedings. The proceedings editors will recommend some high quality papers from the conference to the publication in a special issue of an international journal. This book also includes 4 papers of the special session on High-Performance Data Management and 9 poster papers.

We are delighted to be able to invite Professor Benjamin Wah, University of Illinois at Urbana-Champaign, USA, Professor Amnon Barak, Hebrew University of Jerusalem, Israel, and Professor Wei Zhao, Texas A&M University, USA, as guest speakers. The conference also invited four well-known international scholars for offering two interesting tutorials.

The editors of the conference proceedings would like to take this opportunity to thank all the authors who have submitted their valuable works to this conference for consideration. We thank all the presenters and participants, many of them travel a great distance, for participating in the conference. The editors also thank the PC members and additional technical reviewers for their efforts in reviewing the large number of papers and upholding the high quality of the conference. We greatly appreciate the efforts of the organizers of the special session. Thanks also go to the
members of the organization committee for their consistent supports and many other colleagues who have made this conference possible.

Last but not least we would like to express our gratitude to all of the organizations who support our efforts to bring the conference to fruition. We are grateful to IEEE, Hong Kong Chapter for the cooperation; to The Croucher Foundation for their sponsorship, and to Deakin University and City University of Hong Kong for the sponsorships and organizations.

Welcome to Hong Kong in the best seasons!

Weijia Jia and Wanlei Zhou
Program Committee Co-Chairs
September 1, 2000
CLIENT-SERVER SYSTEMS

By amalgamating computers and networks into a single computing system, a distributed computing system has created the possibility of sharing information and peripheral resources. Furthermore, these systems improve performance of a computing system and individual users through parallel execution of programs, load balancing and sharing, and replication of programs and data. Distributed computing systems are also characterized by enhanced availability and increased reliability.

However, the amalgamation process has also generated some serious challenges and problems. The most important, critical challenge was to synthesize a model of distributed computing to be used in the development of both application and system software. Another critical challenge was to develop ways to hide distribution of resources and build relevant services upon them.

The synthesis of a model of distributed computing has been influenced by a need to deal with the issues generated by distribution such as:
- Locating and accessing remote data, programs, and peripheral resources
- Coordinating distributed programs executing on different computers
- Maintaining the consistency of replicated data and programs
- Detecting and recovering from failures
- Protecting data and programs stored and in transit
- Authenticating users

The model that has been used to develop application and system software of distributed computing systems is the client-server model. Because of this, the current image of computing is client-server distributed computing.

The goal of this article is to introduce and discuss the client-server model and the communication paradigm which gives this model, and to show how this model has influenced the development of different systems and applications.

This article contains three major parts. The first part introduces the client-server model and different concepts and extends to this model. The second part discusses communication supporting distributed computing systems built on client-server model. It contains a detailed discussion of extensions of the communication paradigm: the communication paradigm, one-to-one and group communication, and protocols, message passing and remote procedure call which are used to design and build client-server based systems. The third part presents applications based on the client-server model. The first and simplest presented applications of the client-server model are the network file system (NFS). It is an extension to centralised (local) operating systems (e.g., Unix, MS-DOS) which allows transparent remote file access. Subsequent sections show the RHODOS distributed operating system and distributed computing environment (DCE), respectively. RHODOS has been built from scratch on top of a bare computer. It employs the concept of a microkernel which is a cornerstone of the whole client-server-based operating system. It provides full transparency to the user. On the other hand, DCE is built on top of existing operating systems such as Unix and VMS, and hides differences among individual computers. However, it does not fully support transparency.

THE CLIENT-SERVER MODEL

The Client-Server Model in a Distributed Computing System

A distributed computing system is a set of application and system programs and data dispersed across a number of independent personal computers connected by a communication network. In order to provide requested services to users, the system and relevant application programs must be executed. Because services are provided as a result of executing programs on a number of computers with data stored on one or more locations, the whole activity is called distributed computing.

The problem is how to formalize the development of distributed computing. The main issue of distributed computing is programs in execution, which are called processes. The second issue is that these processes cooperate or compete in order to provide the requested services.

The client-server model is a natural model of distributed computing, which is able to deal with the problems generated by distribution, could be used to describe these processes and their behavior when providing services to users, and allows the design of system and application software for distributed computing systems.

According to this model, there are two processes: the client, which requests a service from another process; and the server, which is the service provider. The server performs the requested service and sends back a response. This response could be a processing result, a confirmation of completion of the requested operation, or even a notice about a failure of an operation.

The client-server model and the association between this model and the physical environment this model is used in are illustrated in Fig. 1. The basic items of the model are the client and server and request and response, and the elements of a distributed computing system are distinguished. This figure firstly shows that the user must send a request to an individual server in order to be provided with a given service. A need for another service requires the user to send a request to another server. Secondly, the client and server processes execute on different computers. They communicate at the virtual (logical) level by exchanging requests and responses. In order to achieve this virtual communication physical messages between these two processes are sent. This implies that operating systems of computers and the communication system of a distributed computing system are actively involved in the service provision.

The most important features of the client-server model are simplicity, modularity, extensibility, and flexibility. Simplicity manifests itself by closely matching the flow of data with the control flow. Modularity is achieved by organizing and integrating a group of computer operations into a separate ser-
Figure 1. The client–server model and its association with operating systems and a communication facility.

In these cases many servers must contribute to the overall application. Furthermore, it would require in some cases simultaneous requests to be sent to a number of servers. Different application will require different semantics for the cooperation between clients and servers.

Distributed computing systems have moved from the basic one-to-one client–server model to the one-to-many and chain models in order to improve performance and reliability. Furthermore, client and server cooperation can be strongly influenced and supported by some active entities which are extensions to the client-server model.

In a distributed computing system there are two different forms of cooperation between clients and servers. The first form assumes that a client requests a temporary service. Another situation is generated by a client which wants to arrange for a number of calls to be directed to a particular serving process. This implies a need for establishing long-term bindings between a client and a server.

Groups in Distributed Computing Systems

A group is a collection of processes, in particular servers, which share common features (described by a set of attributes) or application semantics. In general, processes are grouped in order to deal with this set of processes as a single abstraction: form a set of servers which can provide an identical service (but not necessary of the same quality); encapsulate the internal state and hide interactions among group members from the clients and provide a uniform interface to the external world; and deliver a single message to multiple receivers thereby reducing the sender and receiving overheads (1).

There are two types of groups: closed and open (2). In a closed group only the members of the group can send and receive messages to access the resources of the group. In an open group not only can the members of the group exchange messages and request services but nonmembers of the group can send messages to group members. Importantly, the nonmembers of the group need not join the group nor have any knowledge that the requested service is provided by a group.

Four group structures are often supported to provide the most appropriate policy for a wide range of user application services.
The peer group is composed of a set of member processes that cooperate for a particular purpose. Fault-tolerant and load-sharing applications dominate this type of group style. The client-server group is made up from a potentially large number of client processes with a peer group of server processes. It is an open group. The diffusion group is a special case of the client-server group, where a single request message is sent by client process to all servers. The hierarchical group is an extension to the client-server group. In large applications, with a need for sharing between large numbers of group members, it is important to localize interactions within smaller clusters of components in an effort to increase performance.

According to external behavior, groups can be classified into two major categories: deterministic and nondeterministic. A group is considered deterministic if each member must receive and act on a request. This requires coordination and synchronization between the members of the group. In a nondeterministic group, all members are considered equivalent. Nondeterministic groups assume their applications do not require consistency in group state and behavior, and they relax deterministic coordination and synchronization. Each group member is not equivalent and can provide a different response to a group request, or not respond at all, depending on the individual group member’s state and function.

In order to act properly and efficiently, each member of the group must exchange messages amongst themselves (above group application messages) to resolve the current status of membership of the group. Any change in group membership will require all members to be notified to satisfy the required message requirements. Furthermore, users are provided with primitives to support group membership discovery and group association operations. Group membership discovery allows a process to determine the status of the group and its membership. However, as the requesting process has knowledge of the group members location, a network delay is required.

There are four operations to support group association: create, destroy, join, and leave. Initially a process requiring group association creates the required group. A process is considered to be a group member after it has successfully issued a group primitive, and will remain a member of the group if the process issues a leave group primitive. When the member of the group leaves, the group will be destroyed.

The Three-Tier Client-Server Model

Agents and servers acting as clients can generate different architectures of distributed computing systems. The three-tier client-server architecture extends the basic client-server model by adding a middle tier to support the application logic and common services. In this architecture, a distributed application consists of three components: user interface and presentation processing component, responsible for accepting inputs and presenting the results (the client tier); computational function processing component, responsible for providing transparent, reliable, secure, and efficient distributed computing—it is also responsible for performing necessary processing to solve a particular application problem (application tier); data access processing component, responsible for accessing data stored on external storage devices, such as disk drives (back-end tier).

These components can be combined and distributed in various ways to create different configurations with varying complexity. Figure 2(a) shows a centralized configuration where all the three types of components are located in a single computer. Figure 2(b) shows three two-tier configurations where the three types of components are distributed on two computers. Figure 2(c) shows a three-tier configuration where all the three types of components are distributed on different computers.

Figure 3 illustrates an example implementation of the three-tier architecture. In this example, the upper tier consists of client computers that run user interface processing software. The middle tier consists of computers that run computational function processing software. The bottom tier is back-
end data servers. In a three-tier client–server architecture, application clients usually do not interact directly with the data servers, instead, they interact with the middle tier servers to obtain services. The middle tier servers will then either fulfill the requests themselves, sending the result back to the clients, or more commonly, if additional resources are required, servers in the middle tier will act (as clients themselves) on behalf of the application clients to interact with the data servers in the bottom tier or other servers within the middle tier.

Compared with a normal two-tier client–server architecture, the three-tier client–server architecture demonstrates: (1) better transparency, since the servers within the application tier allow an application to detach the user interface from back-end resources, and (2) better scalability, since servers as individual entities can be easily modified, added, or removed.

**Figure 2.** One- (a), two- (b), and three-tier (c) client–server configurations.

Service Discovery

To invoke a desired service a client must know whether there is a server which is able to provide this service, its characteristics, name, and location. This is the issue of service discovery. In the case of a simple distributed computing system, where there are only a few servers, there is no need to identify the existence of a desired server—information about all available servers is available a priori. This implies that service discovery is restricted to locating the server which provides the desired service. On the other hand, in a large distributed computing system which is a federation of a set of distributed computing systems, with the potential for many service providers who offer and withdraw these services dynamically, there is a need to learn both whether a proper service (e.g., a very fast color printer of high quality) is available at a given time, and if so its name and location. Service discovery is achieved through the following approaches.

**Computer Address is Hardwired into Client Code.** This approach requires the location of the server, in the form of a computer address, to be provided. However, it is only applicable in very small and simple systems, where there is only one server process running on the destination computer.

Another version of this approach is based on a more advanced naming system, where requests are sent to processes rather than to computers. In this case each process is located using a pair (computer address, process name). A client is provided with only the name of a server, but also with the address of a server computer. This solution is not location transparent as the user is aware of the location of the server.

**Broadcast is Used to Locate Servers.** According to this approach each process has a unique name. In order to send a request a client must know the name of the server. However, this is not enough because the operating system of the computer where the server runs must know the address of the server's computer. For this purpose the client’s operating system broadcasts a special locate request containing the name of the server, which will be received by all computers on the network. An operating system which finds the server’s name in the list of its processes sends back a ‘here I am’ reply containing its address (location). The client’s operating system receives the response and can store (cache) the server's name and address.
Computer address for future communication. This approach is
unappealing; however, the broadcast overhead is high as all
computers on a network are involved in the processing of the
request.

Server Location Lookup Is Performed via a Name Server. This
approach is very similar to the broadcast-based approach;
however, it reduces the broadcast overhead. In order to learn
the address of a desired server, an operating system of the
user's computer sends a 'where is' request to a special sys-
tem server, called a name server, asking for the address of a
server where the desired server runs. This means that the
address and location (computer address) of the name server are
sent to all computers. The name server sends back a reply
containing an address of the desired server. The cli-

cient operating system receives the response and can cache
server's computer address for future communication. This
approach is transparent and much more efficient than the
broadcast-based approach. However, because the name server
is centralized, the overall performance of a distributed com-
puting system could be degraded as the name server can be
bottleneck. Furthermore, the reliability of this system is
low; if a name server computer crashed a distributed computing system cannot function.

A Broker is Employed. This approach is very similar to the
server location lookup performed via a name server approach.
However, there are real conceptual differences between a bro-
er and a name server which frees clients from remembering
ASCII names or path names of all servers (and eventually the
server locations), and allows clients to identify attributes of
servers and learn about their availability. A broker is a server
which allows a client to identify available servers which can
be characterized by different attributes describing the
services they provide (e.g., one laser printer is a color printer,
another is a black and white printer). Furthermore, servers
can be offered by some users and revoked dynamically. A user
is not able to know names and attributes of all these servers,
and their dynamically changing availability. There must be a
server which could support users to deal with these problems.
Client–Server Interoperability

Reusability of servers is a critical issue for both users and software manufacturers due to the high cost of software writing. This issue can be easily resolved in a homogeneous environment because the accessing mechanisms of clients may be made compatible with software interfaces, with static compatibility specified by types and dynamic compatibility by protocols.

Cooperation between heterogeneous clients and servers is much more difficult as they are not fully compatible. Thus, the issue is how to make them interoperable. Wegner (5) defines interoperability as the ability of two or more software components to cooperate despite differences in language, interface, and execution platform.

There are two aspects of client–server interoperability: a unit of interoperation, and interoperation mechanisms. The basic unit of interoperation is a procedure (5). However, larger granularity units of interoperation may be required by software components. Furthermore, preservation of temporal and functional properties may also be required.

There are two major mechanisms for interoperation: interface standardization and bridging. The objective of the former is to map client and server interfaces to a common representation. The advantages of this mechanism are: (1) it separates communication models of clients from those of servers, and (2) it provides scalability, since it only requires \( m + n \) maps, where \( m \) and \( n \) are the number of clients and servers, respectively. The disadvantage of this mechanism is that it is closed.

The objective of the latter is to provide a two-way map between client and server. The advantages of this mechanism are: (1) openness, and (2) flexibility—it can be tailored to the requirements of a given client and server pair. However, this mechanism does not scale as well as the interface standardization mechanism, as it requires \( m \times n \) maps.

Conclusions

In this section we introduced the client–server model and some concepts related to this model. Partitioning software into clients and servers allows us to place these components independently on computers in a distributed computing system. Furthermore, it allows these clients and servers to execute on different computers in a distributed computing system in order to complete the processing of an application in an integrated manner. This paves the way of achieving high productivity and high performance in distributed computing.

The client–server model is becoming the predominant form of software application design and operation.

However, to fully benefit from the client–server model, there is a need to employ an operating system and communication network which links computers on which these processes run. Furthermore, in order to locate a server, the operating system must be involved. The question is what class of an operating system can be used. There are two classes of operating systems which could be employed to develop a distributed computing system: a network operating system and a distributed operating system. A network operating system is constructed by adding a module to the local centralized operating system on each computer which allows processes to access remote resources and services; however, in the majority of cases this solution does not fully support transparency.

A distributed operating system is built from scratch, which hides distribution of resources and services; this solution, although futuristic from the current practice point of view, provides location transparency.

It is clear that the extensions to the basic client–server model, described in the previous sections, are achieved through an operating system. Furthermore, network communication services are invoked by an operating system on behalf of cooperating clients and servers.

COMMUNICATION BETWEEN CLIENTS AND SERVERS

Distributed computing systems must be fast in order to instill in users the feeling of a huge powerful computer sitting on or their desks. This implies that communication between the clients and servers must be fast. Furthermore, the speed of communication between remote client and server processes should not be highly different from the speed between local processes. The issue is how to build a communication facility within a distributed computing system to achieve high communication performance.

One of the strongest factors which influences the performance of a communication facility is the communication paradigm: that is, the communication model supporting cooperation between clients and servers and the operating system support provided to deal with the cooperation.

There are two issues in the communication paradigm. Firstly, a client can send a request to either a single server or a group of servers. This leads to two patterns of communication: one-to-one and one-to-many, also called group communication (which are operating system abstractions). Secondly, these two patterns of interprocess communication could be developed based on two different techniques: message passing, adopted for distributed computing systems in the late 1970s; and remote procedure call (RPC), adopted for distributed computing systems in mid-1980s. These two techniques are supported by two respective sets of primitives provided by an operating system. Furthermore, communication between processes on different computers can be given the same format as communication between processes on a single computer.

The following topics are discussed in this section: message passing, including communication primitives; semantics of these primitives; direct and indirect communication; blocking and nonblocking primitives; buffered and unbuffered exchange of messages; and reliable and unreliable primitives are considered. Also, RPC is discussed. The basic features of this technique; parameters, results and their marshalling; client-server binding; and reliability issues are presented.

Thirdly, group communication is discussed. In particular, the basic concepts of this communication pattern; group structures; different types of groups; group membership; message delivery and response semantics; and message ordering in group communication are presented.

Message Passing—Message-Oriented Communication

We define message-oriented communication as a form of communication in which the user is explicitly aware of the message used in communication and the mechanisms used to deliver and receive messages (8).
Basic Message Passing Primitives. A message is sent and received by executing the following two primitives:

\[ \text{send}(\text{dest}, \text{src}, \text{buffer}) \]

The execution of this primitive sends the message stored in buffer to a server process named dest. The message contains a name of a client process named src to be used by the server to send a response back.

\[ \text{receive}(\text{client}, \text{buffer}) \]

The execution of this primitive causes the receiving server process to be blocked until a message arrives. The server process specifies the \text{client} name of a process from whom a message is desired, and provides a buffer to store an incoming message.

It is obvious that the receive primitive must be issued before a message arrives; otherwise the request could be declared as lost and must be retransmitted by the client. Of course, when the server process sends any message to the client process, it must use these two primitives also; the server sends a message by executing the primitive \text{send} and the client receives it by executing the primitive \text{receive}.

There are several points that should be discussed at this stage. All of them are connected with a problem stated as follows: What semantics should these primitives have. The following alternatives are presented: direct or indirect communication via ports; blocking versus nonblocking primitives; buffered versus unbuffered primitives; reliable versus unreliable primitives; and structured forms of message passing based primitives.

Direct and Indirect Communication via Ports. A very basic issue in message-based communication is where do messages go. Message communication between processes uses one of two techniques: the sender designates either a fixed destination process or a fixed location for receipt of a message. The former technique is called direct communication—it uses direct names; the latter is called indirect communication and it exploits the concept of a port.

In direct communication, each process that wants to send or receive a message must explicitly name the recipient or sender of the communication. In this case, the send and receive primitives have the following form: \text{send} \((\text{dest}, \text{src}, \text{buffer})\), \text{receive} \((\text{client}, \text{buffer})\). The \text{dest} and \text{client} are the names of a destination process (server) and sending process (client) from whom the server is prepared to receive a request.

This scheme exhibits a symmetry in naming: that is, both the sender and the receiver have to name one another in order to communicate. A variant of this scheme uses asymmetric naming: only the client names the server, whereas the server is not required to name the client.

Direct communication is easy to implement and to use. It makes a process to control the times at which it receives messages from each process. The disadvantage of the symmetric and asymmetric schemes is the limited modularity of resulting process definition. Changing the name of the server may necessitate the examination of all other processes' definition. All references to the old process must be found, and modified to use the new name. This is not desirable from the point of view of separate compilation. Moreover, receive primitive in a server should allow receipt of a message from any client to provide a service to whatever client process calls it.

Direct communication does not allow more than one client. Similarly, direct communication does not make it possible to send one request to more than one identical server. This implies the need for a more sophisticated technique. Such a technique is based on ports.

A port can be abstractly viewed as a protected kernel object into which messages may be placed by processes and from which messages can be removed: that is, the messages are sent to and received from ports. Processes may have ownership, send, and receive rights on a port. Each port has a unique identification (name) that distinguishes it. A process may communicate with other process by a number of different ports. In this case \text{dest} in the send primitive is a name of a port of the server the request is sent to.

Logically associated with each port is a FIFO queue of finite length. Messages which have been sent to this port but which have not yet been removed by a process reside on this queue. Messages may be added to this queue by any process which can refer to the port via a local name (e.g. capability). A port should be declared. A port declaration serves to define a queueing point for messages. A process which wants to remove a message from a port must have the appropriate receive rights. Usually, only one process may have receive access to a port at a time. Messages sent to a port are normally queued in FIFO order. However, an emergency message can be sent to a port and receive special treatment with regard to queuing.

Blocking versus Nonblocking Primitives. One of the most important properties of message passing primitives concerns whether their execution could cause delay. We distinguish blocking and nonblocking primitives. We say that a primitive has nonblocking semantics if its execution never delays its invoker; otherwise, a primitive is said to be blocking. In the former case, a message must be buffered. The previously described primitives have blocking semantics.

It is necessary to distinguish two different forms of the blocking \text{send} primitive. These forms are generated by different criteria. The first criterion reflects the operating system design and addresses buffer management and message transmission. The blocking and nonblocking send primitives are illustrated in Fig. 4. If the blocking send primitive is used, the sending process (client) is blocked; that is, the instruction following the send primitive is not executed until the message has been completely sent. The blocking receive implies that the process which issued this primitive remains blocked (suspended) until a message arrives, and being put into the buffer specified in the receive primitive. If the nonblocking send primitive is used, the sending process (client) is only blocked for the period of copying a message into the kernel buffer. This means that the instruction following the send primitive can be executed even before the message is sent. This can lead toward parallel execution of a process and message transmission.

The second criterion reflects the client-server cooperation and the programming language approach to dealing with message communication. In this case the client is blocked until the server (receiver) has accepted the request message and
there are two possible solutions: the send may delay until there is space in the buffer for the message, or the send might return a code to the client, indicating that the buffer is full and the message could not be sent.

The situation of the receiving server is different. The receive primitive informs an operating system about a buffer into which the server wishes to put an arrived message. The problem occurs when the receive primitive is issued after the message arrives. The question is what to do with the message. The first possible approach is to discard the message. The client could time out and re-send, and hopefully the receive primitive will be invoked in the meantime. Otherwise, the client can give up. The second approach is to buffer the message in the operating system for a specific period of time. If during this period the appropriate receive primitive is invoked, the message is copied to the invoking server space. If the receive primitive is not invoked, the timeout expires and the message is discarded.

Buffered message passing systems are more complex than unbuffered message passing based systems, since they require creation, destruction, and management of the buffers. Also, they generate protection problems and cause catastrophic event problems when a process owning a port dies or is killed.

Unreliable versus Reliable Primitives. Different catastrophic events, such as a computer crash or a communication system failure, can happen in a distributed computing system. These can cause either a requesting message being lost in the network, a response message being lost or delayed in transit, or the responding computer “dying” or becoming unreachable. Moreover, messages can be duplicated, or delivered out of order. The primitives discussed previously cannot cope with these problems. These are called unreliable primitives. The unreliable primitive send merely puts a message on the network. There is no guarantee of delivery provided and no automatic retransmission is carried out by the operating system when a message is lost.

Dealing with failure requires providing reliable primitives. In a reliable interprocess communication, the send primitive handles lost messages using internal retransmissions and acknowledgments on the basis of timeouts. This implies that when send terminates, the process is sure that the message was received and acknowledged.

Reliable and unreliable receive differ in that the former automatically sends an acknowledgment confirming message reception, whereas the latter does not. Two-way communication requires the utilization of the basic message passing primitives in a symmetrical way. If the client requested any data, the server sends reply messages (responses) using the send primitive. For this reason the client has to set the receive primitive up to receive any message from the server. Reliable and unreliable primitives are contrasted in Figure 6.

Structured Forms of Message Passing Based Communication. A structured form of communication using message passing is achieved by distinguishing requests and replies and providing for bidirectional information flow. This means that the client sends a request message and waits for a response. The set of primitives is as follows.
When remote procedure calls are used a client interacts with a server by means of a call statement

```
      service_name (value_args, result_args)
```

To illustrate that both local and remote procedure calls look identical to the programmer, suppose that a client program requires some data from a file. For this purpose there is a read primitive in the program code.

In a system supported by a classical procedure call, the read routine from the library is inserted into the program. This procedure, when executing, puts the parameters into registers, and then traps to the kernel as a result of issuing a READ system call. From the programmer point of view there is nothing special; the read procedure is called by passing the parameters onto the stack and is executed.

In a system supported by RPC (Fig. 7), the read routine is a remote procedure which runs on a server computer. In this case, another call procedure called a client stub from the library is inserted into the program. When executing, it also traps to the kernel. However, rather than placing the parameters into registers, it packs them into a message and issues the send primitive, which forces the operating system to send it to the server. Next, it calls the receive primitive and blocks itself until the response comes back.

The server's operating system passes the arrived message to a server stub, which is bound to the server. The stub is blocked waiting for messages as a result of issuing the receive primitive. The parameters are unpacked from the received message and a procedure is called in a conventional manner. Thus, the parameters and return address are on the stack, and the server does not see that the original call was made on a remote client computer. The server executes the procedure call and returns the results to the virtual caller: that is, the server stub. The stub packs them into a message and issues send to return the results. The stub comes back to the beginning of its loop to issue the receive primitive, and blocks waiting for the next request message.

The result message on the client computer is copied to the client process (practically to the stub's part of the client) buffer. The message is unpacked, and the results are extracted and copied to the client in a conventional manner. As a result of calling read, the client process finds its data available. The client does not know that the procedure was executing remotely.

It is evident that the semantics of remote procedure calls is analogous to local procedure calls: the client is suspended when waiting for results; the client can pass arguments to the remote procedure; and the called procedure can return results. However, since the client's and server's processes are on different computers (with disjoint address spaces), the remote procedure has no access to data and variables of the client's environment.

There is a difference between message passing and remote procedure calls. Whereas in message passing all required values must be explicitly assigned into the fields of a message before transmission, the remote procedure call provides marshalling of the parameters for message transmission: that is, the list of parameters is collected together by the system to form a message.

---

**e.** Unreliable (a) and reliable (b) message passing primitives.

ud(dest, src, buffer). Sends a request and gets a response; it combines the previous client's send to the server with a receive to get the server's response.

ud_request(client, buffer). Done by the receiver (server) to acquire a message containing work for them to do.

ud_response(src, dest, buffer). The receiver (server) uses this primitive to send a reply after completion of the work.

should be emphasized that the semantics, described in previous sections, can be linked with these primitives. The result of the send and receive combination in the struc-
ture of the send primitive is one operation performed in interprocess communication system. This implies that sending overhead is reduced, buffering is simplified (the request data can be left in a client's buffer, and the use data can be stored directly in this buffer), and the sort-level protocol is simplified.

**e.** Procedure Call

accessing between remote and local processes is visi-
table programmer. It is a completely untyped technique. Transferring message passing based applications is difficult error prone. An answer to these problems is the RPC que which is based on the fundamental linguistic con-
known as the procedure call. The very general term re-
pack procedure call means a type-checked mechanism that is a language-level call on one computer to be automatic-
turned into a corresponding language-level call on an-
computer. The first and most complete description of C concept was presented in Ref. 7.

**ic.** Features of Remote Procedure Calls. The idea of remote procedure calls (RPC) is very simple and is based on the ob-
tion that a client sends a request and then blocks until the server sends a response. This approach is very simi-
lar to conventional programs for centralized computer sys-
tems. It implies that RPC must be transparent. This leads to the main advantages of this communication ap-
bout the programmer does not have to know that the called procedure is executing on a local or a remote computer.
Parameters and Results in RPCs. One of the most important problems of the remote procedure call is parameter passing and the representation of parameters and results in messages. Parameters can be passed by value or by reference. By-value message systems require that message data be physically copied. Thus, passing value parameters over the network is easy: the stub copies parameters into a message and transmits it. If the semantics of communication primitives allow the client to be suspended until the message has been received, only one copy operation is necessary. Asynchronous message semantics often require that all message data be copied twice: once into a kernel buffer and again into the address space of the receiving process. Data copying costs can dominate the performance of by-value message systems. Moreover, by-value message systems often limit the maximum size of a message, forcing large data transfers to be performed in several message operations reducing performance.

Passing reference parameters (pointers) over a network is more complicated. In general, passing data by-reference requires sharing of memory. Processes may share access to either specific memory areas or entire address spaces. As a result, messages are used only for synchronization and to transfer small amounts of data, such as pointers to shared memory. The main advantage of passing data by-reference is that it is cheap—large messages need not be copied more than once. The disadvantages of this method are that the programming task becomes more difficult, and it requires a combination of virtual memory management and interprocess communication, in the form of distributed shared memory.

Marshalling Parameters and Results. Remote procedure calls require the transfer of language-level data structures between two computers involved in the call. This is generally performed by packing the data into a network buffer on one computer and unpacking it at the other side. This operation is called marshalling.

More precisely, marshalling is the process (performed when sending the request as well as when sending the result back) in which three actions can be distinguished:

- Extracting the parameters to be passed to the remote procedure and the results of executing the procedure;
- Assembling these two into a form suitable for transmission among computers involved in the remote procedure call;
- Disassembling them on arrival.

The marshalling process must reflect the data structures of the language. Primitive types, structured types, and user-defined types must be considered. In the majority of cases, marshalling procedures for scalar data types and procedures to marshal structured types built from the scalar ones are provided as a part of the RPC software.

Client-Server Binding. Usually, RPC hides all details of locating servers from clients. However, as we stated in a previous section, in a system with more than one server (e.g., file server, print server), the knowledge of location of clients' files or a special type of a printer is important. This implies the need for a mechanism to bind a client and a server, in particular, to bind an RPC stub to the right server and remote procedure. There are two aspects of binding: the way the client specifies what it wants to be bound to (this is the problem of naming), and the way the client locates the server and the specification of the procedure to be invoked (this is the problem of addressing).

In a distributed computing system there are two different forms of cooperation between clients and servers. The first form assumes that a client requests a temporary service. Another situation is generated by a client which wants to arrange for a number of calls to be directed to a particular serving process. These imply a need for a run-time mechanism for establishing long-term bindings between this client and a server.

In the case of requests for a temporary service, the problem can be solved using broadcast and multicast messages to locate a server. In the case of a solution based on a name server, that solution is not enough, because the process wants
the located server during a time horizon. This means a special binding table should be created containing established long-term binding objects (i.e., a client name and a server name), should be registered. The RPC run-time procedure for performing remote calls expects to be provided a binding object as one of its arguments. This procedure directs a call to the binding address received. It should be possible to add and new binding objects to the table, remove binding objects from the binding table (which in practice means breaking a binding), and update the binding table. In systems with primary servers, broadcasting is replaced by the operation of sending requests to a name server requesting a location of a given server and sending a response with an address of this server. Binding can take place at compile time, link time, or at runtime.

Error Recovery Issues. Because the client and server are separate processes which run on separate computers, they are prone to failures of themselves, their computers, or the communication system. The remote procedure may not be completed successfully. For example, the result message is not returned to the client as a response to its call message, because one of four events may occur: the request message is lost; the result (response) message is lost; the server computer crashes and is restarted; and the client computer crashes and is restarted. These events form the basis for design of RPC recovery mechanisms.

Three different semantics of RPC and their mechanisms are identified to deal with problems generated by these events:

Maybe call semantics. Timeouts are used to prevent a client waiting indefinitely for a response message.

At least-once call semantics. This mechanism usually includes timeouts and a call transmission procedure. The client tries to call the remote procedure until it gets a response or can tell that the server has failed.

Exactly once call semantics. In the case of at least-once call semantics it can happen that the call can be received by the server more than once, because of lost responses. This can have the wrong effect. To avoid this the server sends each time (when retransmitting) as its response the result of the first execution of the called procedure. Thus, the mechanisms for these semantics include, in addition to those used in at least-once call semantics (i.e., timeouts, retransmissions), call identifications and the server's table of current calls. This table is used to store the calls received first time and procedure execution results for these calls.

Message Passing versus Remote Procedure Calls. A problem raises in deciding which of the two interprocess communication techniques is better, if any, and whether there are any suggestions for when, and for what systems, these utilities should be used.

First of all, the syntax and semantics of the remote procedure call are the functions of the programming language being used. On the other hand, choosing a precise syntax and semantics for message passing is more difficult than for RPC because there are no standards for messages. Moreover, neglecting language aspects of RPC and because of the variety of message passing semantics, these two facilities can look very similar. Examples of a message passing system that looks like RPC are message passing for the V system (which in Ref. 8 is called now the remote procedure call system) and message passing for Amoeba (9) and RHODOS (10).

By comparing the remote procedure call and message passing, the former has the important advantage that the interface of a remote service can be easily documented as a set of procedures with certain parameter and result types. Moreover, from the interface specification, it is possible to automatically generate code that hides all of the details of messages from a programmer.

On the other hand, a message passing model provides flexibility not found in remote procedure call systems. However, this flexibility is at the cost of difficulty in the preparation of precisely documented behavior of a message passing interface.

The problem is when these facilities should be used. The message passing approach appears preferable when serialization of request handling is required. The RPC approach appears preferable when there are significant performance benefits to concurrent request handling. RPC is particularly efficient for request-response transactions.

Group Communication

Distributed computing systems provide the opportunity to improve the overall performance through parallel execution of programs on a network of workstations, decreasing the response time of databases using data replication, supporting synchronous distant meetings and cooperative workgroups, and increasing reliability by service multiplexing. In these cases many servers must contribute to the overall application. This implies a need to invoke multiple services by sending a simultaneous request to a number of servers. This leads toward group communication.

The concept of a process group is not new. The V-system (11), Amoeba (2), Chorus (12), and RHODOS (10) all support this basic abstraction in providing process groups to applications and operating system services with the use of group communication.

Basic Concepts of Group Communication. Group communication is an operating system abstraction which supports the programmer by offering convenience and clarity. This operating system abstraction must be distinguished from the message transmission mechanisms such as multicast (one-to-many physical entities connected by a network) or its special case broadcast (one-to-all physical entities connected by a network).

A request is sent by a client called src to a group of servers providing the desired service named group_name by executing either send(group_name, src, buffer) when the message passing technique is used, or call service_name(value_args, result_args) when the RPC technique is used.

This request is delivered following the semantics of a primitive used. The primitives should be constructed such that there is no difference between invoking a single server or a group of servers. This means that communication pattern transparency is provided to the programmer.

Thus, groups should be named in the same manner that single processes are named. Each group is treated as one sin-
gle entity; its internal structure and interactions are not shown to the users. The mapping of group names on multicast addresses is performed by an interprocess communication facility of an operating system and supported by a naming server. However, if multicast or even broadcast is not provided, group communication could be supported by one-to-one communication at the network level.

Communication groups are dynamic. This means that new groups can be created and some groups can be destroyed. A process can be a member of more than one group at the same time. It can leave a group or join another one.

In summary, group communication shares many design features with message passing and RPC. However, there are some issues which are very specific, and their knowledge could be of a great value to the application programmer.

Message Delivery Semantics. Message delivery semantics of a group relates to the successful delivery of a message to processes in a group. There are four choices of delivery semantics:

- **Single Delivery.** Single delivery semantics require that only one of the current group members needs to receive the message for the group communication to be successful.
- **k-Delivery.** In k-delivery semantics, at least k members of the current group will receive the message successfully.
- **Quorum Delivery.** With quorum delivery semantics, a majority of the current group members will receive the message successfully.
- **Atomic Delivery.** With atomic delivery all current members of the group successfully receive the message or none does. This delivery semantic is the most stringent as processes can and do fail and networks may also partition during the delivery process of the request messages, making some group members unreachable.

Message Response Semantics. By providing a wide range of message response semantics the application programmer is capable of providing flexible group communication to a wider range of applications. The message response semantics specify the number and type of expected message responses. There are five broad categories for response semantics:

- **No Responses.** By providing no response to a delivered request message the group communication facility is only able to provide unreliable group communication.
- **Single Response.** The client process expects (for successful delivery of a message) a single response from one member of the group.
- **k-Responses.** The client process expects to obtain k responses for the delivered message from the members of the process group. By using k response semantics the groups resilience can be defined (18). The resilience of a group is based on the minimum number of processes that must receive and respond to a message.
- **Majority Response.** The client process expects to receive a majority of responses from the current members of the process group.
- **Total Response.** The client process requires all current members of the group to respond to the delivery of a request message.

Message Ordering in Group Communication. The semantic of message ordering are an important factor in providing good application performance and reduction in the complexity of distributed application programming. The order of message delivery to members of the group will dictate the type of group it is able to support.

There are four possible message ordering semantics:

- **No Ordering.** This semantic implies that all request messages will be sent to the current group of processes in no apparent order.
- **FIFO Ordering.** This semantic implies that all request messages transmitted in the first-in first-out (FIFO) order by a client process to the current members of the group will be delivered in the FIFO order.
- **Causal Ordering.** The causal ordering semantic delivers request messages to all members of the current group such that the causal ordering of message delivery is preserved. This implies that if the sending of a message a causally follows the delivery of message m, then each process in the group receives m before m*
- **Total Ordering.** Total ordering semantic implies that messages are reliably delivered in sequence to all current members of the group or no member will receive the message. Also, total ordered semantic guarantees that all group members see the same order of message total order is more stringent that FIFO ordering as a message transfers between all current members of the group are in order. This implies that all processes within the current group perceive the same total ordering of messages. In causal ordering we are concerned with the relationship of two messages while in total ordering we are concerned with seeing the same order messages for all group member processes.

Conclusions

In this section we described two issues of the communication paradigm for the client-server cooperation: firstly, the communication pattern, including one-to-one and one-to-many (group communication); secondly, two techniques, message passing and RPC, which are used to develop distributed computing systems. The message passing technique allows clients and servers to exchange messages explicitly using the send and receive primitives. Various semantics, such as direct, indirect, blocking and nonblocking, buffered and unbuffered, reliable and unreliable can be used in message passing. The RPC technique allows clients to request services from servers by following a well-defined procedure call interface. Various issues are important in RPC, such as marshalling and unmarshalling of parameters and results, binding a client to a particular server, and handling exceptions.

SUN'S NETWORK FILE SYSTEM

The first major step in the development of distributed software was made when inexpensive diskless personal computers were connected by inexpensive local networks in order to share a file service or a printer service.
Distributed File Systems

A distributed file system is a key component of any distributed computing system. The main function of such a system is to create a common file system that can be shared by all the clients which run on autonomous computers in the distributed computing system. The common file system should store programs and data and make them available as needed. Since files can be stored anywhere in a distributed computing system, a distributed file system should provide location transparency.

To achieve such a goal, a distributed file system usually allows the client-server model. A distributed file system typically provides two types of services: the file service and the directory service, which are implemented by the file server and the directory server, respectively, distributed over the network. These two servers can also be implemented as a single server. The file server provides operations on the contents of files such as read, write, and append. The directory server provides operations such as directory and file creation and deletion, for manipulating directories and file names. The client application program interface (client API, usually in the form of a process or a group of processes) runs on each client computer and provides a uniform user-level interface for accessing file servers. In this section we will present one of the most important achievements of the 1980s, which is still in use today, the Network File System, known as NFS, developed on the client-server model.

FS Architecture

FS was developed by Sun Microsystems and introduced in 1984 (14). Since then it has been widely used in both industry and academia. NFS was originally developed for use on Unix workstations. Currently, many manufacturers support it for other operating systems (e.g., MS-DOS). Here, NFS introduced based on the Unix system. To understand the architecture of NFS, we need to define the following terms:

NODE. This is a data structure that represents either an open file or directory within the Unix file system. It is used to identify and locate a file or directory within the local file system.

FILEID. The remote file node is a data structure that represents either an open file or directory within a remote system.

DE: The virtual file node is a data structure that represents either an open file or directory within the virtual file system (VFS).

NFS is a virtual file system (linked to VNODEs) that contains all necessary information about the real file system that is managed by the NFS server. A VNODE associated with a given file system is included in a linked list attached to the VFS for that file system.

Server integrates functions of both a file server and an NFS server. The two servers use a uniform NFS/VNODE interface, to access the NFS server. The VFS and VNODE data structures provide the linkage between the abstract uniform file system interface and the real file system (such as a Unix or MS-DOS file systems) that accesses the data. Further, the VFS/VNODE interface abstraction allows NFS to make remote files and local files appear identical to a client program.

In NFS, a client process transparently accesses files through the normal operating system interface. All operating system calls that manipulate files or file systems are modified to perform operations on VFSes/VNODEs. The VFS/VNODE interface hides the heterogeneity of underlying file systems and the location of these file systems. The steps of processing a user-level file system call can be described as follows (Fig. 8):

1. The user-level client process makes the file system call through the normal operating system interface.
2. The request is redirected to the VFS/VNODE interface. A VNODE is used to describe the file or directory accessed by the client process.
3. If the request is for accessing a file stored in the local file system, the VNODE pointed by the VNODE is used. The VNODE interface is used and the request is served by the Unix file system interface.
4. If the request is for accessing a file stored locally in other types of file systems (e.g., MS-DOS file system), a proper interface of that file system is used to serve the request.
5. If the request is for accessing a file stored remotely, the VNODE pointed by the VNODE is used and the request is passed to the NFS client and some RPC messages are sent to the remote NFS server that stores the requested file.
6. The NFS server processes the request by using the VFS/VNODE interface to find the appropriate local file system to serve the request.

The Role of RPC

The communication between NFS clients and servers is implemented as a set of RPC procedures. The RPC interface provided by a NFS server includes operations for directory manipulation, file access, link manipulation, and file system access (15). The actual specifications for these remote procedures are defined in the RPC language, and the data structures used by the procedures are defined in the XDR format. The RPC language is a C-like language used as input into Sun's RPC Protocol Compiler utility. This utility can be used to output the actual C-language source code.

NFS servers are designed to be stateless, meaning that there is no need to maintain information (such as whether a file is open or the position of the file pointer) about past requests. The client keeps track of all information required to send requests to the server. Therefore, NFS RPC requests are designed to completely describe the operation to be performed. Also, most NFS RPC requests are idempotent, meaning that an NFS client may send the same request one or more times without any harmful side effects. The net result of these duplicate requests is the same. NFS RPC requests are transported using the unreliable User Datagram Protocol (UDP). NFS servers notify clients when an RPC completes by sending the client an acknowledgment (also using UDP).
A NFS client sends its RPC requests to a NFS server one at a time. Although a client computer may have several NFS RPC requests in progress at any time, each of these requests must come from a different client. When a client makes an RPC request, it sets a timeout period during which the server must service and acknowledge it. If the server does not acknowledge during the timeout period, the client retransmits the request. This may happen if the request is lost along the way or if the server is too slow because of overloading. Since the RPC requests are idempotent, there is no harm if the server executes the same request twice. If the client gets a second acknowledgment from the request, the client simply discards it.

Conclusions

In this section we showed an application of the client-server model in the development of a distributed file system based on the Network File System. The NFS server integrates functions of both a file server and a directory server. It has been built as an extension module to a centralized operating system (e.g., Unix or MS-DOS). NFS clients use RPC to communicate with the NFS system. This system allows clients running on diskless computers to access and share files.

THE DEVELOPMENT OF THE RHODOS

The vast majority of design and implementation efforts in the area of distributed computing systems have concentrated on client-server-based applications running on centralized operating systems (e.g., Unix, VMS, OS/2). However, there have been huge research efforts on the development of operating systems built from scratch based on the client-server model (called distributed operating systems). These systems support distributed computing systems developed on a set of personal homogeneous computers connected by local or fast wide area networks.

The results achieved have changed, and are still changing operating systems of distributed computing systems and the development of applications supported by these systems. The following systems have been developed based on the client server model: V (8), Amoeba (2), Chorus (12), and RHODO (16).

Distributed Operating Systems

A distributed operating system is one that looks to its user like a centralized operating system, but runs on multiple, independent computers connected by fast local or wide area networks. There are the following four major goals (the first three are the goals of a centralized operating system) of distributed operating system:

- Hide details of hardware by creating abstractions; for example, software which provides a set of higher level functions which form a virtual computer;
- Manage resources to allow their use in the most efficient way and support user processes in the most efficient way;
- Create a pleasant user computational environment; and hide distribution of resources, information, peripheral computational resources, in order to provide full transparency to users.

A generic architecture of a distributed operating system which allows these goals to be achieved has the following three levels. Software providing an abstraction of the hardware, and allows the handling of interrupts and switching. The second level of a distributed operating system is formed by software which manages physical resource...
processor time, memory, input/output, and virtual resources such as processes, remote communication, communication ports, and network protocols. It depends on the support
ded by functions of the software abstraction level. The
second level provides services to the system services level.
Third software level allows the management of files and
tables, service resources, and creates a human user
defined by graphics terminals, command interpreters, and
definition systems. This level creates an image
of the computer system for users. User processes form the soft-
level sitting on the system services level.

A client-server based distributed operating system all,
gement functions and services provided to user pro-
sess are modelled and developed as individual cooperating
processes. User processes act as clients. However, be-
agree in order to achieve the goals of a dis-
operating system. As physicalry in a distributed computing system is not shared, re-
processe communicate using messages. In order to have
form communication model, local processes also commu-
using messages. This provides communication trans-
yness in a natural manner.

This section will use RHODOS to illustrate the appli-
cation of the client-server model in the development of a new
distributed operating systems. For this reason we will
concentrate on the kernel servers and microkernel as
d if new image of operating systems for distributed

Rhodos Architecture

Rhodos (research oriented distributed operating system) is a
kernel and message passing based system developed
the client-server model. This operating system is capa-
adding parallel processing on a network of work-
stations and providing load sharing and balancing in order to
provide high-performance services to users (10).

There are three layers of cooperating processes in
rhodos: user processes, system servers, and kernel servers
(Fig. 9). Each process executes in user mode and is confined
to an individual address space which is controlled and main-
tained by the RHODOS microkernel.

In RHODOS, software creating abstractions forms a mi-
crokernel. The microkernel provides the following functions:
context switching, interrupt handling, basic operations on
memory pages relating to the hardware, and local interpro-
cess communication. Furthermore, this microkernel is respon-
sible for storing and managing basic data structures.

Kernel servers implement the mechanisms of the
RHODOS functionality. Two groups of kernel servers can be
distinguished. To the first group belong the servers which
provide services which could be identified in any distributed
or network operating system: process management, memory
management, remote IPC management, communication pro-
ocols, and I/O management (drivers in RHODOS have also
been developed as individual servers). The second group en-
compases servers which provide advanced services which are
necessary to support parallel processing on a network of
workstations, and load sharing and balancing. These services
are: process migration, remote process creation, and data col-
lection.

System servers implement the policy of the RHODOS func-
tionality. They provide services such as naming, file accessing
and manipulation (in basic and transaction modes), two-way
and m-way authentication, and global scheduling. A broker
service has also been developed and will be installed shortly.
In order to provide these services, system servers act as cli-
ents and invoke relevant kernel servers and the nucleus using
standard system calls.

<table>
<thead>
<tr>
<th>User applications and processes</th>
<th>System servers</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Kernel servers</td>
</tr>
<tr>
<td></td>
<td>The nucleus (microkernel)</td>
</tr>
</tbody>
</table>

![Figure 9. The logical architecture of RHODOS.](image-url)
User applications and processes are those developed and allocated to perform tasks for users. These processes have no special privileges and obtain services by calls to the microkernel and system servers.

Communication in RHODOS

In RHODOS, access to local and remote services is achieved in the same transparent manner, via a system name of that service and uniform interprocess communication, which is provided by the Interprocess Communication (IPC) facility. The facility provides three basic communication primitives: send(), recv(), and call(). Both send() and recv() provide the basic message passing semantics while the call(), recv(), and send() in combination provide synchronous RPC. In providing both message passing and RPC semantics the programmer is able to select the most appropriate communication technique for a given application.

The functioning of the IPC facility is divided into three sections: local IPC module, the IPC manager, and the network manager (Fig. 10). The local IPC module is an integral part of the RHODOS microkernel and provides local communication between processes on the same personal computer. If the destination process exists on the local computer, the module will complete the transfer. Otherwise, the IPC module sends a request to the IPC manager to provide a remote communication service.

The primary responsibility of the IPC Manager is the receiving and transmitting of remote messages for all processes within the RHODOS distributed computing system. It also supports group communication. This service is achieved with the cooperation of the name server by assigning a single name to a group of names. Furthermore, in order to support one-to-one and group communication the IPC manager is responsible for address resolution. In particular, a message that is sent to an individual process or a group requires the IPC manager to resolve the destination processes' (servers') location and provide the mechanism for the transport of the message to the desired process or group of processes.

In order to deliver a message to a remote process (server), the IPC manager invokes a delivery server, called the network manager. This server consists of a protocol stack employing transport, network, and data link protocols.

Currently, the transport service is provided by a fast special- ized RHODOS Reliable Datagram Protocol (RUDP). Network and data link layer protocols are provided by the IP Ethernet suite.

RHODOS Kernel Servers and Services

One of the basic features of the RHODOS design is that each resource is managed by a relevant server: the process manager is responsible for processes and basic operations on processes, the space manager for memory, and the IPC manager for remote group communication and address resolution. A process is a very special resource, because it is constructed based on some basic resources such as space, data structure, usually called process control blocks, communication ports, and buffers. Thus, in RHODOS advanced operations on processes such as process migration and remote process creation are provided by separate servers: the migration manager and REX manager.

Process Manager. The job of the process manager is to manage the processes that are created in RHODOS. The process manager manipulates the process queues and deals with parent processes waiting for child processes to exit. It cooperates with other kernel servers, for instance with the migration manager to transfer a process' state during migration, and the remote execution manager to set up a process' state when a process is created.

Space Manager. One of the goals of RHODOS is portability across hardware platforms. Thus, RHODOS memory management has been separated into two sections: hardware dependent and hardware independent. The small hardware-dependent section is found in the microkernel and the larger hardware-independent section comprises RHODOS space manager. This server deals with spaces, logical units of memory, independent of physical units (e.g., pages), which are mapped to the physical memory.

The space manager supports two types of page operations: copy_on_write, which allows moving processes to share a page while they are reading them but makes separate copies when either process attempts to write to the page copy_on_reference, which is used in process migration when only referenced pages are transferred from a source computer to a destination computer the process has been migrated.

Handling exceptions, creating spaces and transferring pages have been extended by adding additional functions in order to provide an operating system built in support for Distributed Shared Memory (DSM). Two consistency models are supported in the RHODOS DSM: invalidation and update-based.

Device Manager. Transparency is an important feature of RHODOS. This not only includes interprocess communication between remote hosts, but also a transparent unified face of physical devices such as serial ports, keyboards, screens, and disks. Device drivers provide this interface, and vice drivers in RHODOS are in their own right processes with the privilege and status of kernel servers. The benefits can be achieved by implementing device drivers as processes that provide the ability to enable and disable new drivers dynamically, as well as to use normal process debugging tools and functions.
Migration Manager. The process migration manager is responsible for the migration of running processes from one computer to a remote computer. To migrate a process in RHODOS involves migrating the process state, address space, communication state, file state, and other resources. Thus, process migration requires the cooperation of all the servers managing these resources, the process, space, IPC managers, and the file server, respectively. The process migration manager only coordinates these servers, and all of them cooperate following the client-server model.

Process migration in RHODOS is a transaction-based operation performed on processes. Thus, the initial request from the source process migration manager to the destination process migration manager to migrate a selected process starts a transaction. The destination process migration manager commits this transaction by sending a response back, if all operations of installation of resources on the destination computer by individual servers, the process, space, IPC managers, and the file server have been completed successfully. Otherwise, an abort response is sent back.

Remote Execution Manager. The function of the remote execution (REX) manager is to provide coordination for creation processes on local and remote computers. If a process is created on a local computer only a local REX manager is involved. If processes are created on remote computers, the home REX manager cooperates with remote REX managers to ensure processes are created correctly whilst maintaining the link with the process that issued the request. The generic cooperation of the servers is shown in Fig. 11.

Data Collection Manager. The RHODOS Data Collection System is responsible for collection and dissemination of the operational statistics of processes and exchanged messages in the RHODOS environment. The Data Collection System consists of a data collection manager (server) and stubs of code within the microkernel and other servers. The data collection manager is designed to be activated periodically and when special events occur (e.g., a new process was created, a process was killed), and provide a central repository for the accumulation of statistics. It provides accurate process statistics to the global scheduler. These statistics will permit the global scheduler to make the most appropriate decisions concerning process placement within the RHODOS environment.

RHODOS System Servers

The RHODOS system provides direct services to users by employing the following servers: the naming server, file server, authentication server, and the broker server, called the trader. Furthermore, RHODOS provides a special service which improves the overall performance of all services by em-
ploying the global scheduler. The utilization of the client-server model in the development of user-oriented services of a distributed operating system is presented here based on the global scheduler.

RHODOS provides global scheduling services in order to allocate/migrate processes to idle or lightly loaded computers to share computational resources and balance load. Global scheduling employs both static allocation and load balancing. Static allocation is employed when system load remains steady high and new processes have to be created. Static allocation is making the decision of where to create new processes. Load balancing is employed to react to large fluctuations in system load. Load balancing is making a decision when to migrate a process, which process to migrate, and where to migrate this process. These servers make these decisions based on the information about the current load of the personal computers participating in global scheduling, their load trends, and the process communication pattern.

Conclusions

In this section, we showed an application of the client-server model in the development of an advanced distributed operating system, RHODOS. RHODOS consists of a microkernel and two layers of cooperating servers, called kernel servers and system servers. Generally speaking, kernel servers implement the mechanism of the RHODOS functionality, whereas system servers implement the policy of the RHODOS functionality. User processes, sitting on top of the RHODOS software, obtain services from RHODOS servers. When a RHODOS server receives a service request, it may serve the request directly, or it may contact other servers if services from these servers are required.

BUILDING THE DISTRIBUTED COMPUTING ENVIRONMENT ON TOP OF EXISTING OPERATING SYSTEMS

The previous section contains a presentation of RHODOS, an example of a distributed operating system, developed based on the client-server model and the concept of a microkernel. The whole system has been built from scratch on bare hardware. There is another approach to building a distributed computing environment by putting it on top of existing operating systems. Such a software layer hides the differences among the individual computers, and forms a single computing system.

The Role of the Client-Server Model in Building a Distributed Computing Environment

Open Software Foundation's Distributed Computing Environment (DCE) (17) is a vendor-neutral platform for supporting distributed applications. DCE is a standard software architecture for distributed computing that is designed to operate across a range of standard Unix, VMS, OS/2, and other operating systems. It includes standards for RPC, name, time, security, and thread services—all sufficient for client-server computing across heterogeneous architectures.

DCE uses the client-server model to support its infrastructure and transparent services. All DCE services are provided through servers. By using DCE, application programmers can avoid considerable work in creating supporting services, such as creating communication protocols for various parts of a distributed program, building a directory service for locating those pieces, and maintaining a service for providing security in their own programs.

In the previous section, we mainly addressed the kernel servers and microkernel of RHODOS as they are the result of the new approach based on the client-server model and the concept of a microkernel to building distributed computing systems. Here, since DCE is a complete extension of centralized operating systems to form a distributed computing system, we mainly concentrate on servers which directly provide services to users.

The Architecture of DCE

The architecture of DCE masks the physical complexity of the networked environment by providing a layer of logical simplicity, composed of a set of services that can be used separately or in combination to form a comprehensive distributed computing system. Servers that provide DCE services usually run on different computers; so do clients and servers of a distributed application program that use DCE.

DCE is based on a layered model which integrates a set of fundamental technologies (Fig. 12). To applications, DCE appears to be a single logical system with two broad categories of services (18):

The DCE Core Services. They provide tools with which software developers can create end-user applications and system software products for distributed computing:

- *Threads.* DCE supports multithreaded applications;
- *RPC.* The fundamental communication mechanism which is used in building all other services and applications;
- *Security Service.* Provides the mechanism for writing applications that support secure communication between clients and servers;
- *Cell Directory Services (CDS).* Provides a mechanism for logically naming objects within a DCE cell (a group of client and server computers);
- *Distributed Time Service (DTS).* Provides a way to synchronize the clocks on different computers in a distributed computing system.
DCE Data-Sharing Services. In addition to the core services, DCE provides important data-sharing services, which require no programming on the part of the end user and which facilitate better use of shared information:

Distributed File Service (DFS). Provides a high-performance, scalable, secure method for sharing remote files;

Enhanced File Service (EFS). Provides features which greatly increase the availability and further simplify the administration of DFS.

In a typical distributed environment, most clients perform their communication with only a small set of servers. In DCE, computers that communicate frequently are placed in a single cell. Cell size and geographical location are determined by the people administering the cell. Cells may exist along social, political, or organizational boundaries and may contain up to several thousand computers. Although DCE allows clients and servers to communicate in different cells, it optimizes the more common case of intra-cell communication. One computer can belong to only one cell at a time.

The Role of RPC

DCE RPC is based on the Apollo Network Computing System NCA/RPC). The components of DCE RPC can be split into the following two groups according to the stage of their usage:

Used in Development. It includes IDL (Interface Definition Language) and the idl compiler. IDL is a language used to define the data types and operations applicable to each interface in a platform independent manner. idl compiler is the tool used to translate IDL definitions into code which can be used in a distributed application;

Used in Runtime. It includes RPC runtime library, rpcd (RPC daemon), and rpcp (RPC control program).

To build a basic DCE application, the programmer has to apply the following three files:

The Interface Definition File. It defines the interfaces (data structures, procedure names, and parameters) of the remote procedures that are offered by the server;

The Client Program. It defines the user interfaces, the calls to the remote procedures of the server, and the client side processing functions;

The Server Program. It implements the calls offered by the server.

C E uses threads to improve the efficiency of RPCs. A thread is a lightweight process that executes a portion of a program, operating with other threads concurrently executing in the same address space of a process. Most of the information that a part of a process can be shared by all threads executing within the process address space. Sharing reduces significantly the overhead incurred in creating and maintaining a connection, and the amount of information that needs to be saved when switching between threads of the same

The Servers of DCE

All the higher-level DCE services, such as the directory services, security service, time service, and distributed file services, are provided by relevant servers.

Directory Services. The main job of the directory services is to help clients find the locations of appropriate servers. To let clients access the services offered by a server, the server has to place some binding information into the directory. A directory is a hierarchically structured database which stores dynamic system configuration information. The directory is a realization of the naming system. Each name has attributes associated with it, which can be obtained via a query using the name.

Each cell in a DCE distributed computing system has its own directory service, called the Cell Directory Service (CDS), that stores the directory service information for a cell (18). It is optimized for intra-cell access, since most clients communicate with servers in the same cell. Each CDS consists of CDS servers and CDS clients. A CDS server runs on a computer containing a database of directory information (called the clearhouse). Each clearhouse contains some number of directories, analogs to but not the same as directories in a file system. Each directory, in turn, can logically contain other directories, object entries, or soft links (an alias that points to something else in CDS).

Each cell may have multiple CDS servers. Nodes which do not run a CDS server must run a CDS clerk. A CDS clerk acts as an intermediary between a distributed application and the CDS server on a node not running a CDS server.

When a server wishes to make its binding information available to clients, it exports that information on one of its cell's CDS servers. When a client wishes to locate a server within its own cell, it imports that information from the appropriate CDS server by calling on the CDS clerk on its computer.

DCE uses the Domain Name System (DNS) or Global Directory Service (GDS, based on the X.500 standard) to enable clients to access servers in foreign cells. To access a server in a foreign cell, a client gives the cell's name and the name of the desired server. A CDS component called a Global Directory Agent (GDA) extracts the location of the named cell's CDS server from DNS or GDS, then a query is sent directly to this foreign server.

Security Service. DCE provides the following four security services: authentication, authorization, data integrity, and data privacy. A security server (it may be replicated) is responsible for providing these services within a cell. The security server has the following three components:

Registry Service. It is a database of principal (a user of the cell), group, and organization accounts, their associated secret keys, and administration policies.

Key Distribution Service. It provides tickets to clients. A ticket is a specially encrypted object that contains a conversation key and an identifier that can be presented by one principal to another as a proof of identity.

Privilege Service. It supplies the privileges of a particular principal. It is used in authorization.
The security server must run on a secure computer, since the registry on which it resides contains a secret key, generated from a password, for every principal in the cell. They are based on the Kerberos V5.0, created by the MIT/Project Athena, and DCE extends Kerberos version 5 by providing authorization services.

**Time Service.** Distributed Time Service (DTS) of DCE is designed to keep a set of clocks on different computers synchronized. DTS uses the usual client-server structure: DTS clients, daemon processes called clarks, request the correct time from some number of servers, receive responses, and then reset their clocks as necessary to reflect this new knowledge.

There are several components that compose the DCE DTS:

- **Time Clerk.** It is the client side of DTS. It runs on a client computer and keeps the computer's local time synchronized by asking a time server for the correct time and adjusting the local time accordingly.

- **Time Servers.** There are three types of time servers. The local time server maintains the time synchronization of a given LAN. The global time server and counter time servers are used to synchronize time among interconnected LANs. A time server synchronizes with other time servers by asking these time servers for correct times and by adjusting its time accordingly.

- **DTS API.** It provides an interface where application programs can access time information provided by the DTS.

**Distributed File Services.** DCE uses its distributed file services (DFS) to join the file systems of individual computers within a cell into a single file space. A uniform and transparent interface is provided for applications to accessing files located in the network. DFS is derived from the Andrew File System. It uses RPC for client-server communication and threads to enhance parallelism; it relies on the DCE directory to locate servers; and it uses DCE security services to protect against attackers.

DFS is based on the client-server model. DFS clients, called cache managers, communicate with DFS servers using RPC on behalf of user applications. There are two types of DFS servers: a fileset location server which stores the locations of system and user files in DFS, and a file server which manages files.

A typical interaction between various components of DFS is shown in Fig. 13. At first, the application issues a file request call to the cache manager in its computer. If the requested file is located in the local cache, the request is served using the local copy of the file. Otherwise, the cache manager locates the fileset location server through the CDS server, and the location of the file server that stores the requested file is found through the fileset location server. Finally, the cache manager calls the file server and the file data are accessed.

**Conclusions**
In this section we described an application of the client-server model in the development of an advanced distributed computing environment, DCE. DCE is built on top of existing operating systems and it hides the heterogeneity of underlying computers by providing an integrated environment for distributed computing. DCE consists of many integrated services, such as thread and RPC services, security service, directory service, time service, and distributed file service, that are necessary in performing client–server computing in a heterogeneous environment. Most of these services are implemented as individual servers or groups of cooperating servers. Application processes act as clients of DCE servers. New in its fifth year (DCE 1.0 was announced in 1991), DCE has gone through several major stages of evolution and enhancement (through DCE 1.1 and DCE 1.2). Because of its operating system independence, DCE has gained significant support from user and vendor communities.

**BIBLIOGRAPHY**


ANDRZEJ GOSCINSKI
WANLEI ZHOU
Deakin University
INTEGRATING HETERogeneous DATABASES: A DISTRIBUTED MODEL

P. A. HEPNER, W. ZHOU
School of Computing and Mathematics
Deakin University
Geelong, VIC 3217, Australia
E-mail: {philh, wanlei}@deakin.edu.au

In this paper we put forward a distributed model for accessing heterogeneous database systems. The processing of database operations requested by the user are performed in a distributed manner that takes advantage of the inherent parallelism of distributed systems, minimises network traffic and uses almost any general purpose computer on the network. Processing is not confined to DBMS sites but is provided as a distributed service. The design is modular and provides the mechanisms that may be arranged in a variety of ways providing a range of integration paradigms from loosely coupled integrations to more tightly coupled integrations.

1 Introduction

The proliferation of heterogeneous database systems has continued for many years. Small systems developed to meet the needs of individuals, departments or divisions have appeared. These systems are usually designed for a specific need by external developers that specialise in that area of activity. The need to integrate these disparate systems has led to heterogeneous distributed database systems.

Current database integrating systems are single site as with gateways, middle-ware and other three tiered systems, or they are a common interface system which is popular in today's client centric systems. They attempt to centralise control and processing that creates a situation where there is a reliance and dependency on single entities in a distributed environment. This reliance often leads to a system that is less reliable where reliability is an often claimed benefit of distributed systems. Centralisation is an activity that is some what contrary to the whole notion of distributed systems.

As heterogeneous databases are almost always distributed, as are the users, our aim is to develop a distributed database integration model that is flexible, scalable and modular. This model will act as a precursor to the construction of a system in order to investigate the practicalities and performance issues relating to a distributed database integration system.

To achieve this, chapter two introduces the model and discusses the issues relating to it design including database model, environmental issues, parallel processing, resource optimisation, and operator heuristics. Chapter three
describes the proposed model and architecture outlining the components and their interaction. Chapter four concludes the paper.

2 Design Issues

Our model uses the relational database model as the underlying database model. Relational database manipulation may be expressed as a relational algebraic expression which may also be represented graphically as a hierarchy, called a relational algebraic tree, with the root node being the result, or user view, and the leaves being the base relations. Intermediate nodes perform the relational operations.

In our model, for each relational algebraic tree we instantiate a process for each node in the tree and provide pair wise inter process communication between nodes. Inter process communication may be local or remote and, as will be seen in Section 3, processes may be grouped at one computer or distributed to achieve some degree of optimisation.

The root and leaves of the tree are tied to the location of the user process and component database tables respectively. Intermediate nodes may be placed between the former two in a manner that takes advantage of geographical layout of the network, parallelism and the heuristic nature of the operations being performed. Processing nodes will not be bound to the component database or integrating sites. The inter process communication between all processing entities is a well defined client/server protocol. Each instantiated node in the tree is a client to the server below it, and visa versa, and the protocol used is an abstraction of a relational table. Figure 1 is an example of an execution strategy for the relational algebraic tree. Here a user issues the query on Computer A. The information required by the query are stored in Computers E and G, respectively. Computers A, B, C, D, E and F perform various tasks of the query processing.

2.1 Geography

An aspect of process placement that should be considered is the geographical nature of the network, the location of component database systems and the user application. The root and leaves are tied to certain locations for each instance of a tree but the intermediate nodes could be placed anywhere the scheduling policy desires given there is a mechanism to support this.

To facilitate this the entity that is to schedule and place the relational operator processes will need a map of the network and nodes. Processing nodes are classified as passive or active with respect to their ability to provide relational operation processing. Passive nodes are the routers, bridges and
switches of the network and provide network segmentation but are unable to provide operator processing. Active nodes have the ability to execute relational operations and are the more general purpose computers of the network.

The intention of this is that the execution tree may be instantiated by overlaying it onto the network map. Relational operations will be performed by the active nodes and the majority of data will flow from the leaves to the root with processing along the way as shown in the example in figure 1. Note that the sensible position for the join operation in this environment is where it is placed in the diagram as the fork in the tree matches the fork in the network.

A dynamic map creation and maintenance technique appears to be a more reliable mechanism reducing dependencies. This would enable nodes capable of processing relational operations to make themselves available and unavailable as desired by systems or users. Failure would have less of an effect on the system but there would be some communication overhead involved.

2.2 Parallel Processing and Pipelining

Distributed systems are inherently parallel in that each processor may execute in parallel and independently from other processors. To take advantage of this the processes in a relational algebraic tree may be distributed as shown in Figure 1. As many relational operators process one row at a time each process may work independently on separate rows. Once completed a process would pass the processed row to the next process closer to the root and obtain a new row or rows from the process or processes closer to the leaves. Such processing may be likened to a pipeline of processes used in the UNIX operating system...
to manipulate rows of text.

The granularity of processing is a product of the algorithms used in providing the operation. The granularity is either row at a time or entire table processing. Row at a time retrieval may be by a particular order, retrieval via a key or retrieval via an index. Only row at a time processing is suitable to pipelining.

We note that some branches of the tree may support pipelines better than others. This again relates to the algorithms used to perform the operations and for many operations more than one algorithm may be used to perform the same task giving different pipelining options. The investigation of the algorithms and their pipeline ability will be a substantial part of future research.

2.3 Resource Optimisation

One of the general aims of this distributed model is to improve performance, but to do this we must provide optimisation techniques. Optimisation involves finding some balance between the utilisation of resources to achieve better performance and less load.

We define the two dominate resources in distributed systems as being processors and networks. Resource users, being processes and inter process communication, are to be allocated to the respective resource in a manner that attempts to avoid the over use of a resource.

By placing two or more communicating processes on the same computer we are able to reduce network traffic but increase the load on the processor. By spreading the processes across computers we are able to share the processing load but at the cost of increased network communication. This gives us the flexibility to trade off the load of one resource to reduce the load of another.

3 The Architecture

To achieve a distributed execution strategy defined in the previous section we use use a number of system processing entities. They are as follows:

3.1 Component Table Access Server

A component table access server (CTAS) represents a table that a component database is willing to share with other databases and applications. A CTAS is as close to a component database tables as this system gets and represent component database tables as a canonical client/server interface. CTAS's are instantiated as the leaves in the execution strategy of an algebraic tree and pro-
vide the homogenising layer to what may otherwise be heterogeneous database tables.

3.2 Common Operation Servers

Common Operation Servers (COPS's) include relational database operations and non-relational operations. They provide common operations that are frequently used by other entities and applications of the system.

Relational database operations are positioned as non-leaf nodes in the execution strategy. There are two types of filtering COPS's that represent the relational select and project operators. They filter out rows and columns respectively. An instantiated filtering COPS is a client to another COPS and is therefore a node within an execution tree.

Other relational operators perform an aggregation process. An instantiated COPS of this type is a client to two other COPS's and therefore forms a fork in the execution tree. The aggregation of the two client tables with relational operators such as join, union, intersection, etc. forms a view of a single table on the server side. The ability to update may or may not be blocked.

Non-relational operations provide operations that are not part of the relational model but are needed to provide functionality required by users and/or the system. User activities such as sorting fall into this category. System functions include activities such as mapping and translation that are required when there is a need for name or type conflict resolution.

3.3 Database Access Points

A database access point (DAP) provides access to databases via a well defined client/server protocol. The DAP provides the following:

- User authentication. The user requests access to a given database. This is the access control mechanism and until the user is authorised he/she has access to only one database table, the list of databases this DAP controls. Once authorised to access the database requested, the user would also have access to a table that is a list of tables. Depending on the functionality of the DAP, each user or group of users may have their own database view.

- Table access. The user requests access by presenting a relational algebraic expression, in its simplest form the request would simply state a table name. The DAP schedules the expression and provides the user with a client/server connection to the defined table which may be local or
remote. Tables presented by the DAP may be defined as a stored view (see below). The scheduling process will graft the users expression to the stored view expression and optimise it before instantiating the strategy.

Component Database Access Point

All CTAS's of each component database participating in an integration can be accessed via a Component Data Access Point (CDAP). Regardless of the heterogeneity of the component database systems the CDAP provides a consistent, or canonical interface in the form of a client/server protocol. The primary objective of the CDAP is to provide access to relations (tables) represented by CTAS's of the component database. In its simplest form the CDAP would present the relations as they are with minimal conversion, only that required to meet the requirements of the canonical client/server protocol. Name and type mapping may be performed by the integrating system above this or the CDAP may perform the operations itself. It is likely that at least some conversion or mapping is required.

A CDAP has the ability to instantiate the homogenising processes that enable heterogeneous tables to be manipulated with the canonical interface.

Integrating Database Access Point

An integrating data access point (IDAP) will have knowledge of two or more CDAP's. An IDAP may also use other IDAP's as its component DAP's which would form a hierarchy of IDAP's. In Sheth and Larson's terminology the IDAP represents the federating schema.

In its simplest form the IDAP's list of tables would be derived by the union of the tables listed at the CDAP's it is integrating. This is similar to many multi-database implementations in that the schema is controlled by the component database systems. This has the benefit of minimising the dependencies created by storing references to component database entities at the IDAP. Changes to component database structure is automatically reflected in the integrating systems meta-data.

The disadvantage of this simple mechanism is that the mapping of data types and names, and the semantic matching and aggregation of data items between component databases is left to the user. The user is required to have knowledge of the component database semantics so data items may be matched in a meaningful manner. A more tightly coupled integration would have the integration supported by the mapping of data type, name, value etc. through the use of views that are stored as relational algebraic expressions at the IDAP. The IDAP needs not have any local base tables as it may reference them at
other DAP's via the use of views. This implies the only stored information required at the IDAP is a table of view definitions.

The stored views create a dependency problem but enable a better control over the data. Mechanisms such as replication and fragmentation between component databases are able to be supported through the use of special non-relational operators.

When a user process, such as a query language processor, requests a view in terms of IDAP tables, the scheduling system will graft the tree produced by the requesting process to those defined by the views stored at the IDAP, and so on down the hierarchy. Once grafted and optimised, this tree is instantiated giving the user process the required result.

3.4 Agents

Each DAP has the responsibility of setting up a distributed tree of processes for each table or view accessed by a user. To facilitate this we introduce the agent process. Each node or computer that is willing to participate acts as an agent. When an agent becomes available it will broadcast within its domain its willingness to participate. In this way it may be included in a map of the domain at any DAP. If the scheduler at a DAP decides to place one or more CTAS's or COPS's at the location of the agent it will send a message to the agent requesting their instantiation. The message will include the subsection of the tree to be instantiated, the parameters, if any, for each CTAS, or COPS, and communication connection information.

Agents are independent of the integration system using them. Two or more completely independent database integrations may use the same agents. This makes agents a pool of shared resources available to any DAP that wishes to use them.

4 Conclusion

The model for database integration presented in this paper differs from other designs in number of ways. It assumes the component database systems and user processes are distributed and attempts to take advantage of this. It does so by distributing the processing in a manner that utilises the inherent parallelism of the distributed system while attempting to minimise the network traffic with the general aim of providing improved performance. This gives the system the potential to out perform current homogeneous distributed database systems by taking advantage of the geographical layout of the network and the inherent parallelism of distributed systems.
The system separates the mechanism from the strategy so that a variety of process scheduling schemes and topologies may be used. Redundant points of access may be provided by this system avoiding reliance on a central site.

Finally the design provides a modular approach to database integration in that it provides the mechanisms that may be arranged in a variety of ways. This enables the user to choose the type of integration that suits his or her needs at the time of implementation. The choice between loosely coupled and more tightly coupled database integration's is an open choice and organisations may mix different degrees of coupling as they choose. Such a choice is not provided by other systems as they have been designed around a particular problem and lack the modularity and flexibility to do so.

References


Efficient Algorithms for Mobile Multicast using Anycast Group

Weijia Jia, Wanlei Zhou, Joerg Kaiser, and M. A. Livani

Department of Computer Science, City University of Hong Kong, 83 Tat Chee Ave, Hong Kong
School of Computing & Mathematics, Deakin University, 662 Blackburn Rd, Melbourne, Australia
Department of Computer Structures, University of Ulm, James-Franck-Ring, 89069 Ulm, Germany

ABSTRACT

This paper presents a novel and efficient multicast algorithm that aims at reducing delay and communication cost for the registration between mobile nodes and mobility agents and solicitation for foreign agent services based on Mobile IP. The protocol applies anycast-addressing technology to support multicast transmissions for both mobile nodes and home/foreign agents. Mobile hosts use anycast tunneling to connect to a set of nearest available home/foreign agent where an agent is able to use anycast to route the multicast message to the nearest multicast router for efficient multicasting of messages. The performance analysis and experiments demonstrate that the algorithms are able to enhance the performance over existing remote subscription and bi-directional tunneling approaches.

1 INTRODUCTION

Multicast is an important service for mobile applications through wireless connection to the Internet, such as email communication, query database, retrieve information, video conferencing and resource discovery through wired network etc. The provision of multicast service to mobile nodes is a complex task especially in the wireless environment. The physical constraints of mobile communications typically include low bandwidth of link layer connection, high error rates, and temporary disconnection.

In mobile multicast communications, two issues are primary important: One is for mobile nodes and mobility agents to discover each other’s presence and another is the datagram routing efficiency. Current Mobile IP [9] uses similar approach of ICMP [12] for mobility agent discovery in which an agent periodically multicast or broadcast the agent advertisements to those links and mobile nodes for offering routing services. Mobile nodes also use broadcast to connect mobility agents even a mobile node is in the home network. Mobile computing requires wireless communication, mobility and portability. Typically, a wireless link between the mobile nodes and the point of attachment has a low bandwidth, high error rate and poor connectivity. Thus, the protocols for mobile nodes should rely more computation and communication power on those static hosts instead of mobile nodes. Due to dynamic group membership and location change of mobile nodes, reliable and efficient mobile multicast design and implementation are even more challenging.

This paper proposes a novel efficient mobile multicast protocol/algorithms (called MMP) that adapts anycast technology originally proposed by IPv6 [13]. It intends to reduce the time for the agent discovery as associated with existing approaches based on ICMP. MMP uses approaches of efficient anycast tunneling and anycast routing technology for group registration and multicast delivery across the environment of mobile connection to Internet. The contribution of this paper is that for those mobile agents that provide connection for mobile nodes, efficient registration can be achieved through connection with anycast group. Thus mobile nodes may use well-known address to connect to a “nearest” mobility agent for registration. In this way the connection time can be reduced substantially.

The rest of the paper is organized as follows. Section 2 illustrates the relevant issues of mobile multicast, its architecture and algorithms. Section 3 presents the MMP algorithms. Section 4 describes the performance analysis and simulation results. Section 5 concludes the paper with some discussions.

2 RELATED WORK

Multicast is a communication that involves a single sender and multiple receivers. Traditional multicast research discussed ordering and reliability of message delivery in the multicast group in guaranteeing the properties such as total ordering, atomicity, dynamic group membership and fault-tolerance etc [14]. IP multicast [5] provides unreliable multicast delivery for wired networks. Several well-known multicast routing protocols have been developed as extensions to the existing routing protocols. Distance-Vector Multicast Routing Protocol (DVMRP) [1] is an extension to Routing Information Protocol (RIP). MOSPF [2] is a multicast extension to the Open Shortest Path First (OSPF), which is a link state protocol. Cored Based Tree (CBT) [3] is a shared tree routing algorithm as shown in Figure 1 (we will discuss the issue later). Protocol Independent Multicast (PIM) [4] is designed to be independent of underlying unicast routing protocol.

Mobile IP [9], in essence, has a way of doing three relatively separate functions:

1. Agent discovery - Home agents and foreign agents (HA and FA) may advertise their availability on each link for which they provide service. A newly arrived mobile node can send a solicitation on the link to learn if any prospective agents are present. A mobility agent may transmit agent advertisements to advertise its services on a link. Mobile nodes use those advertisements to determine their current point of attachment to the Internet.

- 423 -
A mobility agent is required to limit rate at which it sends broadcast or multicast agent advertisements with the maximal rate once per second [16].

![Multicast Spanning Tree](image)

**Figure 1. A Multicast Spanning Tree**

2. **Registration** — When a mobile node is away from home, it registers its care-of address with its home agent. A care-of address is an IP address at the mobile node's current point of attachment to the Internet, when the mobile node is not attached to the home network. Depending on its method of attachment, the mobile node will register either directly with its home agent or through a foreign agent, which forward the registration to the home agent.

3. **Tunneling** — In Mobile IP and IP multicast, unicast tunnels are used to encapsulate and to send multicast packets over the Internet when the intermediate routers cannot handle multicast packets. In order for multicast datagrams to be delivered to the mobile node when it is away from home, the home agent has to tunnel the datagrams to the care-of address. A mobile node is addressed on its home network that is called home address.

There have been some well-known wireless multicast systems developed. Forwarding pointers and location independent addressing to support mobility has been discussed in [11], but the multicast service is unreliable. Host View Management Protocol (HVMP) provides reliable multicast for mobile nodes [12]. However, it does not allow dynamic group membership. Reference [13] proposed a protocol that allows dynamic group changes and reliable multicast message delivery with different network architecture. Multicast tunneling is proposed to forward multicast packets from one foreign network to another when the mobility agent receives packets addressed to mobile nodes that are nomadic [8]. Perkins [9] defined two approaches to support mobile multicast, which are called remote subscription and bi-directional tunneling multicast. The remote subscription is inefficient for dynamic membership and location change of mobile nodes. Bi-directional tunneling multicast may cause the tunnel convergence problem with packet duplication [10] (see Figure 2).

![Bi-directional Tunneled Multicast Method](image)

**Figure 2. Bi-directional Tunneled Multicast Method**

Anycast address and anycast service have been defined for Internet Protocol version 6 (IPv6) [13]. Anycast is a communication for a single sender sends to the “nearest” member in a group of receivers, preferably only one of the servers that support the anycast address [11]. It uses unicast address space. A router can register the anycast address for its interface similar to multicast address registration. Anycast can simplify the task of finding an appropriate server when a host requests a service from a server in a group but does not care which server is used.

3 OUR PROTOCOL

This section discusses a Mobile Multicast Protocol (MMMP) that provides efficient multicast services for mobile nodes by employing anycast agent group approach. To form an agent group, an anycast address is configured by a group of mobility agents (both home/final agents) on the subnet that are designed to support a specific multicast group. When a registration request is sent to the anycast address, a home agent, upon reception of request, can follow the same method used in Mobile IP and send a registration reply to the mobile node. The anycast group approach targets two purposes: 1) Using a well-known anycast address, the home agents and mobile nodes may register directly through well-known anycast address of the home/mobile anycast agent groups. This approach will reduce the time and cost for advertisement of home agents as well as for the solicitation of mobile nodes. 2) Using anycast can dynamically select the paths to the multicast router, as a result reducing the end-to-end multicast delay. The second issue has been elaborated and detailed in references [2, 18].

3.1 Assumptions and Notations

Before proceeding the description of the protocol, the following assumptions are made:

1. A set of hosts and mobile nodes forms a multicast group. Each individual mobile node has knowledge of the multicast group id to which it is interested in transmission and reception of multicast messages.

2. Both home agent and foreign agents (denoted as Mobility Agent-MAs) are special routers that provide service for the attachment of mobile nodes.

3. The Internet environment consists of interconnected sub-nets. There is at least one MA in each sub-net.
4. Multicast routers can configure its interface to route both multicast and anycast packets (refer to next subsection).

We introduce the following notations for the protocol design:
1. $G$ - A multicast group/address and it is also taken as the group id or address such as IP class D multicast address.
2. $G_a$ - An anycast address/group registered by all mobility agents that provide multicast service for group $G$.
3. $T_a$ - An anycast address/group registered by all routers that link the members in $G$.
4. $ML(G)$ - Membership list maintained by a MA, it contains the ids of members in group G.
5. $VL(G)$ - Visitor-list maintained by a MA, it is used to record the ids of foreign mobile nodes that belong to $G$.
6. $AL(G)$ - Away-list maintained by MA, it is used to record the ids of mobile nodes in $G$ that already left away from the home MA.
7. $TL(G)$ - Tunnel list maintained by MA, is used to record the ids of foreign agents that have tunneling connections to this MA and are interested in transmission/reception of multicast packets of $G$.

The functions applied to the lists:
1. $id(host/MA)$ - return the identity of the host/node or an agent;
2. $Insert(id, L)$ - insert identity $id$ into the list $L$;
3. $Move(id, L_1, L_2)$ - move id from list $L_1$ to $L_2$ where $L_1 \neq L_2$.

3.2 Design of MMP

The idea for the protocol is roughly described below. In the network, a group of routers join together to provide multicast transmission for a group of (mobile) nodes. MMP uses CBT technology [4] to establish a multicast tree to connect the routers and nodes (in case a node is a mobile node, its home agent is on the leaf of the tree). The MMP devises two anycast groups for the efficient multicast communications. The mobility agents in each subnet that provide multicast delivery for group $G$ form the anycast group with reserved anycast address $G_a$ [19]. The multicast routers (MR) on the multicast tree form an anycast group and use a $T_a$ as the temporary anycast address. MASs may select any one of MR for tunneling the multicast packets for the service of mobile nodes throughout the lifetime of group $G$. Therefore, in the heterogeneous mobile and Internet environment, the leaves of multicast tree are mobility agents and fix host whereas the mobile nodes are taken as the dynamic leaves. MMP is divided into three major phases:

1. **Initialization phase**: Multicast and anycast Group formations for routers, mobility agents and mobile nodes.
2. **Registration and membership phase**: Registration and reformation for the dynamic membership of mobile nodes.
3. **Multicast transmission phase**: Multicast packet transmissions and deliveries for the hosts and mobile nodes.

Phase 2 and Phase 3 works interactively. They are discussed in details in the next subsections.

**Initialization phase**

This phase constitutes of four steps:

**Step 1, Membership Initialization**: An individual MA that provide multicast services for group $G$ initiates $ML(G)=VL(G)=AL(G)=TL(G)=1$.

**Step 2, Multicast Tree Formation**: CBT techniques are used to build a multicast propagation tree for the routers (called a CBT tree). One router is selected as the core (or root) of the tree. To establish such a tree, Mobility Agents that provide multicast service for $G$ join the CBT tree by linking itself to the core (see [4] for details). Note that all routers including MASs that are along the tree to the core are called on-tree routers.

**Step 3, Mobility Agent Anycast Group Configuration**: The mobility agents that offer attachment for the mobile node in $G$ form an anycast group [17]. All the mobility agents can register any group id (address), here denoted as $G_a$. The agent that registered through group $G_a$ may configure one of its interfaces to accept the registration of home or foreign mobile nodes. Address $G_a$ can be made as a well-known address through the network under the concern of $G$. In our algorithm, we propose a concept association agents for anycast group $G_a$. For two agents $MA_1$ and $MA_2$ where $MA_1 \in G_a$ and $MA_2 \in G_a$, then $MA_1$ and $MA_2$ are called association agents and denote as $MA_1 \leftrightarrow MA_2$. If $MA_1 \leftrightarrow MA_2$, then $MA_1$ and $MA_2$ agree on the authentication and multicast packet delivery, i.e., $MA_1$ and $MA_2$ allow each other to delegate the multicast service and agree on the attachment of the mobile node that previously attached to another party. The mobility agents in the same anycast group are called in the same association.

**Step 4, On-tree Router Anycast Group Configuration**: For the group $G$, virtual anycast address $T_a$ is assigned to and configured by all routers related to group $G$ [18]. The routers are classified as on-tree routers and off-tree routers. They configure their interface for $T_a$ in different ways:

- **On-tree Router Configuration**: For a multicast group $G$, when the CBT tree is built, all on-tree routers (including the core) on the tree are selected to join an anycast group with anycast address $T_a$ which is advertised to the network (broadcast by the core). $T_a$ may be considered as a "temporary" anycast address as long as the CBT tree exists. For these nodes (hosts) outside the tree, their attaching router is not the on-tree router, may assign $T_a$ as an interface entry and their routing table is configured with $<T_a, G>$ mappings. For any on-tree router, there is a Forwarding Information Base (FIB) used as its multicast routing table [4, 17]. An entry of the FIB has the form of $<G, input-interface, output-interfaces>

- **Off-tree Router Configuration**: Upon reception of address $T_a$ broadcast from the core in CBT tree, the off-tree routers, including foreign agents, that are interested in transmitting multicast service for $G$ will register $T_a$ on the anycast address registration should be similar to multicast address.
its routing table. The anycast routing table enables the router to dynamically select a “better” path to reach the CBT tree among multiple paths even in the presence of link or next hop failure. About details of fault-tolerant CBT routing algorithms, we refer interested readers to references [18].

Registration and Dynamic Membership Phase
With the proposed anycast group approach, a mobile node may learn the existing agents by caching the anycast address through DHCP or SLP services [16, 20]. In the register message of mobile node, normally the D-bit is set to enable the mobile node to receive/de-capussulate incoming multicast packet [9]. MMP allows membership changes to be made to a anycast group G. A mobile node is allowed to join or leave a anycast group at free. The concept of dynamic group membership is similar to the host view and supervisor host [7]. To join the multicast group G in the home network, a mobile node must register through the home agent. In current Mobile IP, a mobility agent must also broadcast advertisement messages periodically (similar to ICMP advertisement messages) and the mobile node has to send solicitation message to contact with the agent when it hears no advertisement for a certain period of time. In the following, we will describe how the MMP is designed to reduce the cost of advertisement using anycast group. The phase is divided into several sub-phases as below:

Sub-phase I. Mobile Node Registration: A mobile node Mn can register through its home agent and join G for multicast message transmission. The registration can be accomplished through the anycast connection technique. Assume that a mobile node knows the anycast address of Ga for those agents that provide multicast services of G. Mn may use address Ga to connect with the “nearest” MA in its home network. Because we have assumed that in each sub-net, there exists at least one MA, therefore, the solicitation procedure terminates eventually. Upon establishment of the connection between MN and MA, two cases must be considered:

Case 1: The MA is an on-tree router for group G and executes the following steps:
1. Similar to Mobile IP [9], the MA performs the similar corresponding authentication and mobility binding such as care-of address assignment to Mn etc.
2. Insert(id(Mn), ML(G)).

Case 2: The MA is an off-tree router for group G. Similar to Case 1, the MA must first check authentication of Mn. After the MA accepts the connection request from Mn, it learns the group id G through Mn, it initiates ML=G=I and calls Insert(id(Mn), ML(G)).

Following sub-cases must be resolved:

• Sub-case 1: The MA is a multicast router and it uses Ga to join the CBT tree for G by sending join-request to anycast address Ta. Note that Ta has been used as the common interface for the multicast routers in the CBT tree. In this way, it links itself to the CBT tree. Note that the join-request is sent to the “nearest” on-tree router with address of Ta [17, 18].

• Sub-case 2: The MA is not a multicast router. It builds an anycast tunnel to the “nearest” on-tree router so that a single “tree trunk” is grafted on the CBT tree (see [18] for detail).

Sub-phase 2. Mobile Node Visit a Foreign Network: When a mobile node visits a foreign sub-net, it has to send a register request with information of its home agent address, home address and care-of address to the foreign agent (FA) in another network [9]. It also informs the FA about the id of multicast group G from which it is interested in receiving/sending multicast packets. As the current form of Mobile IP [9], foreign agents seldom broadcast the advertisements for the presence of attachment service. The mobile nodes should first contact to a foreign agent with registration request and then the FA connects to the home agent for authentication of the mobile node. On successful reception of registration acknowledge message from the home agent, the FA replies to the mobile node with “acceptance” messages and adds Mn into its group membership list ML by calling Insert(id(Mn), GL(G)).

With the concept of anycast group association, the registration cost for a mobile node on the foreign network can be reduced considerably. Suppose the mobile node Mn originally registered in MA1 in net 1 and now is migrating to foreign network net 2 with MA2. Consider two cases:

Case 1: MA2∈Ga, by definition of association, since both MA1 and MA2 are in Ga, i.e., they are in the same authentic group. Mn may use address Ga to contact with MA2 for registration. MA2, upon checking authentication and acceptance for Mn, executes Insert(id(Mn), VL(G)).

On the other hand, MA1 calls Move(id(Mn), ML(G), AL(G)).

Case 2: MA2∉Ga, MA2 does not provide service for multicast group G. Thus, MA2 applies bi-directional tunneling approach similar to Mobile IP. Mn first contacts MA2 with registration. MA2, upon reception of the registration request, contacts with MA1 for authentic check etc. Upon acceptance the visit of Mn, MA2 calls Insert(id(Mn), VL(G)). Since MA2 is not an on-tree router, it sets a tunnel to MA1 to receive the multicast packets and make them delivery for Mn. On the other hand, MA1 calls Insert(id(MA1), TL(G)) to record the tunneling information. The dynamic tunneling approach grafts MA2 on the multicast tree temporarily.

Sub-phase 3. Mobile Node Leaves: When a mobile node leaves its home network, it notifies its home agent MA by sending a deregistration message. The later calls Move(id(Mn), ML(G), AL(G)). In case both ML(G)=I and TL(G)=I, i.e., the MA does not have any mobile node attached to G nor any tunnel to members in G, then MA uses IGMP message to notify its up-link node until core to trim this branch from the CBT tree [13].

Multicast transmission phase
During the message transmission, if a mobile node is using a co-located care-of address, it should use this address as the source IP address of its IGMP [12] messages; otherwise, it is required to use its home address.
Multicast transmission: A mobile node may generate a multicast message \( m \), intending to send to \( G \). Message \( m \) is thus transmitted to home agent HA. When MA receives \( m \), it first encapsulates \( m \) with a multicast header and then embeds \( m \) within an anycast address \( T_a \) into an anycast packet \( m_a \). The packet is then routed to the address \( T_a \) using dynamic anycast routing algorithms (refer to [17] for details). When a router in \( T_a \) receives the anycast packet, it strips off the anycast header of \( m_a \), inserts \( m \) and propagates it across group \( G \). For a visited mobile node \( M_G \), if it wants to send the multicast packet, the packets can be forwarded through the foreign agents. Like Mobile IP, a co-located care-of address on the foreign network is required and used as the source address for multicast packets to group \( G \).

Multicast packet reception and delivery: When a MA receives an encapsulated multicast packet \( m \) from a router on CBT tree, it strips off the anycast header from the packet and sends the packet delivery to the ids in \( ML(G) \) and \( VL(G) \). The packet is also tunneled to the agents in \( TL(G) \).

4 PERFORMANCE

This section presents the performance analysis for the MMP protocol, particularly, compares the complexity of MMP with remote subscription (RS) and bi-directional (BD) approach in terms of number of broadcast/multicast packets sent and delay introduced. We will demonstrate experiment results to show availability of the protocol by simulation results.

4.1 Analysis

To analyze the performances of the MMP protocol, we use following metrics for the comparison of MMP with methods proposed in Mobile IP [9]:

- **Number of Messages (m/bcasts)** – the number of messages (including multicast and broadcast) required for certain operations such as registration etc.
- **Delay** – total delays in seconds to accomplish a operation. \( \Delta \) is used to measure a single multicast/broadcast (minimum) transmission delay in seconds.

<table>
<thead>
<tr>
<th>Operations</th>
<th>Protocols</th>
<th># of Msgs (m/bcasts)</th>
<th>Delay (sec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent Discovery</td>
<td>Mobile IP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MMP</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Registration</td>
<td>RS</td>
<td>2</td>
<td>1+2( \Delta )</td>
</tr>
<tr>
<td>on HA</td>
<td>MMP</td>
<td>2</td>
<td>2( \Delta )</td>
</tr>
<tr>
<td>Registration</td>
<td>BD</td>
<td>4</td>
<td>1+4( \Delta )</td>
</tr>
<tr>
<td>on FA</td>
<td>MMP</td>
<td>2</td>
<td>2( \Delta )</td>
</tr>
</tbody>
</table>

**Table 1. Performance Comparisons**

According to Mobile IP, the agent discovery requires the MA to send broadcast for agent advertisement. Mobile nodes use these advertisements to determine their current point of attachment to the Internet. The advertisement is sent at a rate of once every second (so the delay). Therefore, for a mobile node, it has to wait for the advertisement and then learns the presence of mobile agent.

With MMP, making use of anycast address \( O_A \), mobile nodes have the knowledge of presence of MA. The cost of agent advertisement can be saved.

For registration of a mobile node, we differentiate the registration on the home agent (HA) from that on the foreign agent (FA). If the registration is on HA, MMP uses the similar number of messages as mobile IP. But delay is shorter as MMP does not wait for the advertisements of HA. Only the transmission delay is taken into account. Mobile IP makes use of bi-directional tunneling for a mobile node to registrar to a foreign network under the assumption that its home agent is a multicast router. The mobile node tunnels IGMP messages to its home agent and then the home agent forwards multicast datagram down the tunnel to the mobile nodes. It is known that four messages are required: a mobile node sends the request to FA, then FA relays the request to HA. HA, in turn, sends back a message of acceptance or deny to FA and then FA relays the final status to the mobile node. While in MMP, if the FA is in the same anycast group as that of FA, only two messages are required: the registration through FA is the same as through HA. For the delay analysis, the reason is similar to the above argument.

4.2 Simulation

In the simulation, it is assumed that there are \( N \) local area networks with \( M \) mobile nodes. For simplicity, we only consider the performance simulations on the mobile nodes. Each LAN has two mobility agents (i.e. one home agent and one foreign agent).

We assume one multicast group and \( M \) multicast routers. It is also assumed that each multicast group has only one source for generating multicast messages in ratio of \( \alpha \) time-units. To simplify the simulation, the location of LAN is fixed for the duration of each simulation. For example, a simulation runs with \( N = 16 \) LANs has a 4-4 coordinate system. The network topology is not drawn for simplicity. We execute 10 simulations for each set of workload parameters and achieve the mean value. The experiment compares the effectiveness of multicast delivery of our protocol to bi-directional tunneling in terms of message delivery delay and number of delivered messages. Figure 3 presents simulation results with \( N = 9 \), one group, simulation time = 255s, and generated multicast messages = 8,500.

Figure 3 (a) shows that our protocol can provide a better multicast service to mobile nodes as the message delivery delay is lower than that of bi-directional tunneling. The high delay demonstrates the transmission overhead in the tunnel from home network to foreign network of bi-directional tunneling. Figure 3 (b) shows that about 90% of the generated messages were delivered to the mobile nodes by MMP and about 50% of the generated message were delivered by bi-directional tunneling protocol. For MMP, two situations may affect the delivery of multicast messages to the mobile nodes: (1) the node may be in transit and (2) the node may be attached to a network with poor link connection due to bad environments. The unsuccessful
deliveries in bi-directional tunneling may be caused by inconsistent information in home network about the location of its mobile nodes.

**Figure 3.** (a) Message delivery delays. (b) Number of delivered messages.

## 5 CONCLUSIONS

MMP extends Mobile IP with anycast address group technology for agent discovery, registration of mobile nodes and delivery of multicast packets. The utilization of anycast address for the mobility agent group can reduce the cost and delay when the mobile nodes make registration with mobility agents among the subnets without impacting its performance. In contrast to bi-directional tunneling and remote subscriptions, MMP is more efficient in terms of delivery delay and throughput of multicast packets. The cost of the employing anycast address/group is comparable to the management of multicast addresses, thus will not compromise the dynamic performance of MMP. MMP is a preliminary research to provide an extension for Mobile IP with efficiency for mobile node registration and connections when multicast services are desired. There are many rooms remain for further research of using anycast group such as the issues of the multiple multicast groups for multiple agents in a subnet and dynamic anycast group management etc.

**REFERENCES**

Efficient algorithm for mobile multicast using anycast group

W. Jia, W. Zhou and J. Kaiser

Abstract: The authors present a novel and efficient multicast algorithm that aims to reduce delay and communication cost for the registration between mobile nodes and mobility agents and solicitation for foreign agent services based on the mobile IP. The protocol applies anycast group technology to support multicast transmissions for both mobile nodes and home/foreign agents. Mobile hosts use anycast tunnelling to connect to the nearest available home/foreign agent where an agent is able to forward the multicast messages by selecting an anycast route to a multicast router so as to reduce the end-to-end delay. The performance analysis and experiments demonstrated that the proposed algorithm is able to enhance the performance over existing remote subscription and bidirectional tunnelling approaches regardless of the locations of mobile nodes/hosts.

1 Introduction

Mobile computing requires wireless communication, mobility and portability. Mobile multicast [1] is an important service for mobile applications through wireless and connection to the Internet, such as email communication, query database, retrieving information, video conferencing through wired networks etc. The provision of a multicast service to mobile nodes is a complex task especially in the wireless environment. The physical constraints of mobile communications typically include low bandwidth of link layer connection, high error rates, and temporary disconnection. IP multicast [2] provides unreliable multicast delivery for wired networks. In mobile multicast communications, two issues are of primary importance: one is for mobile nodes and mobility agents to discover each other’s presence, and another is the datagram routing efficiency. Traditional multicast research discussed reliability of message delivery in the multicast group in guaranteeing properties such as total ordering, atomicity, dynamic group membership and fault-tolerance etc. [3].

Some well known wireless multicast systems have been developed. Forwarding pointers and location-independent addressing to support mobility has been discussed [4], but the multicast service is unreliable. A host view management protocol (HVMP) has been developed that provides reliable multicast for mobile nodes [5]. However, it does not allow dynamic group membership. Brown and Singh [6] have proposed a protocol that allows dynamic group changes and reliable multicast message delivery with different network architectures. Multicast tunnelling is proposed for forwarding multicast packets from one foreign network to another when the mobility agent receives packets addressed to mobile nodes that are nomadic [7].

1.1 Problems with mobile IP

Mobile IP [1] defined three approaches to support mobile connection and multicast: first, agent discovery, where home agents (HAs) and foreign agents (FAs) may advertise their availability on each link for which they provide service. A newly arrived mobile node can send a solicitation on the link to learn if any prospective agents are present. Secondly, remote subscription, when a mobile node is away from home, it registers its care-of-address (an IP address at the mobile node’s current point of attachment to the Internet) when it is not attached to the home network [1] with its home agent. Depending on its method of attachment, the mobile node will register either directly with its home agent or through a foreign agent, which will forward the registration to the home agent. Thirdly, bidirectional tunnelling multicast, in this case unicast tunnels are used to encapsulate and to send multicast packets over the Internet when the intermediate routers cannot handle multicast packets. For multicast datagrams to be delivered to the mobile node when it is away from home, the home agent has to tunnel the datagrams to the care-of address. A mobile node is addressed on its home network that is known as its ‘home address’. Agent discovery may require more advertisements and solicitation messages. Remote subscription is inefficient for dynamic membership and location change of mobile nodes. Bidirectional tunnelling multicast may cause the tunnel convergence problem with packet duplication [5] (Fig. 1).

Fig. 1 Bidirectional tunnelling multicast method

FA – foreign agent; FN – foreign network; HA – home agent; HN – home network; MH – mobile host

© IEE, 2001

IET Proceedings online no. 20010211
DOI: 10.1049/ip-Com:20010211
Paper first received 9th August 1999 and in revised form 24th July 2000

W. Jia is with the Department of Computer Science, City University of Hong Kong, 33 Tat Chee Avenue, Hong Kong

W. Zhou is with the School of Computer Science and Mathematics, Deakin University, 662 Blackburn Road, Melbourne, Australia

J. Kaiser is with the Department of Computer Structures, University of Ulm, James-Franck-Str., 80069 Ulm, Germany


509
1.2 Motivation of the research
The anycast address and service have been defined for Internet protocol version 6 (IPv6) [7]. It is a communication for a single sender sending to the 'nearest' member in a group of receivers, preferably only one of the servers that supports the anycast address [8]. It uses a unicast address and the router can register the anycast address for its interface. Anycast is useful when a host requests a service from a server in a group but does not care which server is used. Anycast can simplify the task of finding an appropriate server. For example, users can use the anycast address to choose the mirrored FTP sites and to connect to the nearest (available) server.

To improve the efficiency in terms of mobile IP on multicast communication, particularly in terms of the three issues mentioned above, we propose a novel efficient mobile multicast protocol (MMP), taking advantage of anycasting routing technology. The MMP has two aims: first, mobility agents (MAs, both HAs and FAs) anycast group to facilitate flexible connections for mobile nodes. Using a well known anycast address, the HAs need not multicast/broadcast router advertisement and the mobile nodes may register directly through the well known anycast address of the anycast agent groups so as to reduce the connection cost for the mobile nodes. Secondly, an anycast address is configured by a group of multicast routers on the subnet that are designed to support a specific multicast group. Using anycast can dynamically select the paths to the multicast router to reduce the end-to-end multicast delay. The second issue has been considered in [9, 10] and we omit the discussion in this paper due to lack of space.

2 Our mobile multicast protocol (MMP)
Before describing the protocol, the following assumptions are made (see Fig. 2 for example of MMP topology):

- A set of hosts and mobile nodes forms a multicast group G. Each individual mobile node has knowledge of the multicast group id to which it wishes to transmit and accept multicast messages.
- HA and FA are special routers that provide service for the attachment of mobile nodes.
- There is at least one MA in each subnet.

A multicast router can configure its interface to route both multicast and anycast packets [9]. Each MA maintains four lists for the dynamics of mobile nodes in multicast group G: the membership list, \( ML(G) \), contains the IDs of members in group G; the visitor list, \( VL(G) \), records the IDs of foreign mobile nodes that belong to G that visit this MA; the away list, \( AL(G) \), records the IDs of mobile nodes in G that departed (or were disconnected) from this MA; finally, the tunnelling list, \( TL(G) \), records the IDs of foreign agents that are interested in transmission/reception of multicast packets for G. The MMP is designed in three major phases that work interactively:

(i) Initialisation phase: configurations of multicast and anycast group for routers, mobility agents and mobile nodes;
(ii) Registration and membership phase: registrations and reformation for the dynamic membership of mobile nodes;
(iii) Multicast transmission phase: multicast packet transmissions and deliveries for the group of members including station hosts and mobile nodes.

2.1 Phase 1: Initialisation

(i) Membership initialisation for a given group G: an individual MA sets \( ML(G) = VL(G) = AL(G) = TL(G) = \{ \} \).

(ii) Multicast tree formation: The core-based tree (CBT) technique is used to build a multicast propagation tree for the routers (called a CBT tree). One router is selected as the core (or root) of the tree. To establish such a tree, MAs that provide multicast service for G must join the CBT tree by linking themselves to the core [10, 11]. All routers including MAs in the tree are called ontree routers.

(iii) Mobility agent anycast group configuration: the mobility agents that offer attachment for mobile nodes in G form an anycast group [9]. All the mobility agents that provide connections for G can register through well known group reserved anycast address \( G_x \) [8] and configure one of its interfaces to accept the registration for home/foreign mobile nodes. Our protocol defines that the agents in the same anycast group \( G_x \) will share the same authentication for mobile node registrations, i.e. \( MA_1 \in G_x \) and \( MA_2 \in G_x \) imply that both \( MA_1 \) and \( MA_2 \) agree to delegate connection authentication and multicast packet delivery to each other for the mobile nodes that were previously attached to another party.

(iv) Ontree router anycast group configuration: for the group G, virtual anycast address \( T_y \) is assigned to and configured by all routers in the CBT tree for group G [9]. The router configurations are classified as ontree and offtree:

- **Ontree router configuration:** For a multicast group G, when the CBT tree is built, all ontree routers (including the core) are selected to join an anycast group with anycast address \( T_y \) which is advertised to the network (broadcast by the core). \( T_y \) may be considered as some 'temporary' anycast address as long as the CBT tree exists. For any ontree router, there is a forwarding information base (FIB) used as its multicast routing table [9, 11]. An entry in the FIB has the form \( <G, input-interface, output-interfaces> \).

- **Offtree router configuration:** Upon reception of address \( T_y \) broadcast from the core in the CBT tree, the offtree routers, including those foreign agents, that are interested in transmitting multicast packets to G will assign \( T_y \) as an interface entry by configuring with \( <T_y, G> \) mappings in the routing table. The anycast routing table enables the router to dynamically select a 'better' path to reach the CBT tree among multiple paths even in the presence of link or hop failure. For details of fault-tolerant CBT routing algorithms, we refer interested readers to [10].
2.2 Phase 2: Dynamic member registration and connection

With the proposed anycast group, a mobile node may learn existing agents by caching the anycast address through DHCP or SLP services [12, 13]. In the register message of mobile node, normally the D-bit is set to enable the mobile node to receive/decapsulate incoming multicast packet [1]. MMP allows membership changes to be made to a multicast group G. A mobile node is allowed to join or leave a multicast group at will. The concept of dynamic group membership is similar to the host view and supervisor host [14]. To join a multicast group G in the home network, a mobile node must register through the home agent. In the current mobile IP, a mobility agent must also broadcast advertisement messages periodically (similar to ICMP advertisement messages [15]) and the mobile node has to send a solicitation message to contact the agent when it hears no advertisement for a certain period of time. This phase is designed to reduce the cost of advertisement using an anycast group by the following steps:

Step 1. Mobile node home registration: A mobile node Mn must register through its home agent and join G for multicast transmission. The registration can be accomplished through anycast connection by using G, to connect to the 'nearest' MA in its home network. On establishment of the connection between MA and Mn, two cases must be considered:

Case 1: The MA is an ontree router of G. Similar to the mobile IP [1], the MA performs the corresponding authentication and mobility binding such as care-of address (CA) assignment to Mn (denoted as CA(Mn)) and calls Insert(CA(Mn), ML(G)) to insert CA of Mn into membership list ML(G).

Case 2: The MA is an offtree router. Similar to case 1, the MA must first check authentication of Mn, then calls Insert(CA(Mn), ML(G)). The following subcases must be considered:

- Subcase 1: The MA is a multicast router and uses G, to join CBT tree for G by sending join-request to the 'nearest' ontree router in Td [9, 10].
- Subcase 2: The MA is a multicast router. It builds an anycast tunnel to the 'nearest' ontree router so that a single 'tree trunk' is grafted on the CBT tree [10].

Step 2. Mobile node visits a foreign network: A mobile node Mn originally registered in MA, subnets 1 and moves to foreign network subnet 2 to connect with MA2. Two cases must be considered:

Case 1: MA2 ∈ G, since both MA1 and MA2 are in G, they are in the same authentication group. Mn may use address G, to make contact with MA2 for registration. On checking authentication and acceptance for Mn, MA2 executes Insert(CA(Mn), VL(G)). On the other hand, MA1 calls Move(CA(Mn), ML(G), AL(G)) to move CA of Mn from the membership list ML(G) to the away list AL(G).

Case 2: MA2 ∉ G, MA2 does not provide service for multicast group G. Thus, MA2 applies a bidirectional tunnelling approach similar to the mobile IP [1]. Upon acceptance of the visit of Mn, MA2 calls Insert(CA(Mn), VL(G)). Since MA2 is not an ontree router, it sets a tunnel to MA1 and the later calls Insert(tos(MA1), TL(G)) to record the tunnelling information for MA2.

Step 3. Mobile node leaves: When a mobile node leaves its home network, it should notify its home agent by sending a deregistration message. The latter calls Move(CA(Mn), ML(G), AL(G)). If ML(G) = VL(G) = TL(G) = 1, i.e. the MA has neither a mobile node attached to G nor any tunnel for visitor members in G, then the MA uses an IGMP message to notify its up-link node until 'core' to trim this branch from the CBT tree [16].

Step 4. Foreign mobile node leaves: An MA may set up a specific timeout for the foreign mobile node in list VL(G). When the timer expires, the MA just deletes the node ID from its VL(G). A similar approach can be applied for the management of list TL(G).

2.3 Phase 3: Multicast transmission phase

(i) Multicast transmission: A mobile node may generate a multicast message m intending to send to G. Message m is thus transmitted to home agent MA. When MA receives m, it first encapsulates m with a multicast header and then imbeds m within an anycast address Ta into an anycast packet p. The packet is then routed to the address Ta using dynamic anycast routing algorithms [9]. When a router in Ta receives the anycast packet, it strips off the anycast header of p into m and propagates it across group G. For a visited mobile node Mn, if it wants to send the multicast packet, the packets can be forwarded through the FAs. As in the mobile IP, a co-located care-of address on the foreign network is required and used as the source address for multicast packets to group G.

(ii) Multicast packet reception-delivery: When an MA receives an encapsulated multicast packet p from a router on the CBT tree, it strips the multicast header from the packet and makes the packet delivery to the IDs in ML(G) and VL(G). The packet is also tunnelled and retransmitted to the agents in TL(G) when TL(G) is not empty.

Note that if the mobile node is using a co-located care-of address, it should use this address as the source IP address of its IGMP [16] (membership) messages; otherwise, it is required to use its home address for multicast transmissions.

3 Performance

This section presents the performance analysis for the MMP protocol and demonstrates experimental results to show the availability of the protocol by simulation results. In particular, it compares the complexity of MMP with remote subscription (RS) and bidirectional (BD) approaches in terms of number of broadcast/multicast packets and end-to-end delay of multicast.

Table 1: Performance comparisons

<table>
<thead>
<tr>
<th>Operations</th>
<th>Protocols</th>
<th>Number of messages (m/locasts)</th>
<th>Delay (s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Agent discovery</td>
<td>Mobile IP</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>MMP</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>Registration on HA</td>
<td>RS</td>
<td>2</td>
<td>1+2Δ</td>
</tr>
<tr>
<td></td>
<td>MMP</td>
<td>2</td>
<td>2Δ</td>
</tr>
<tr>
<td>Registration on FA</td>
<td>BD</td>
<td>4</td>
<td>1+4Δ</td>
</tr>
<tr>
<td></td>
<td>MMP</td>
<td>2</td>
<td>2Δ</td>
</tr>
</tbody>
</table>

3.1 Analysis

To analyse the performances of the MMP protocol, we use the following metrics for the comparison of MMP with methods proposed in mobile IP [1]:

- number of messages (m/locasts), this is the number of messages (including multicast and broadcast) required for the corresponding operation.
* delay, this is the total delays in seconds to accomplish the operation and $\Delta$ is used to measure a single multicast/broadcast (minimum) transmission delay.

According to the mobile IP, the agent discovery requires the MA to send a broadcast for agent advertisement. Mobile nodes use these advertisements to determine their current point of attachment to the Internet. The advertisement is sent at a maximum rate of once every second (hence the delay). Therefore, for a mobile node, it has to wait for the advertisement and then it discovers the presence of MA. With MMP, in the presence of anycast address $G_a$, mobile nodes are aware of the presence of MA. Thus no agent advertisement is required.

For registration of a mobile node, we differentiate the registration on the HA from that on the FA. If the registration is on the HA, in terms of message number, MMP is the same as the protocols based on mobile IP. But the delay is shorter as MMP does not wait for the advertisements of HA. Only the transmission delay of two messages is taken into account.

Mobile IP makes use of bidirectional tunnelling for a mobile node to register to a foreign network under the assumption that its HA is a multicast router. The mobile node tunnels IGMP messages to its HA and the HA forwards the multicast datagram down the tunnel to the mobile nodes. It is known that four messages are required: one is the request from a mobile node to FA, then FA relays the request to HA. HA, in turn, sends back a message of acceptance or denial to FA and then FA relays the final status to the mobile node. While in MMP, if the FA is in the same anycast group as that of the HA, only two messages are required: the registration through FA is the same as through HA. For the delay analysis, the reasoning is similar to the above argument.

3.3 Simulation results
The experiment compares the effectiveness of multicast delivery of MMP with bidirectional tunnelling in terms of message delivery delay and number of delivered messages. The simulation considers one multicast group with up to 90 (mobile) nodes across nine LANs, and 8500 multicast messages are generated within 2500s.

![Fig. 4 Message delivery delays](image)

N = 9, 8500 messages generated
- - - - MMP
- - - - bidirectional tunnelling

![Fig. 5 Number of delivered messages](image)

N = 9, 8500 messages generated
- - - - MMP
- - - - bidirectional tunnelling

Fig. 4 shows that our protocol can provide a better multicast service to mobile nodes as the message delivery delay is lower than that of bidirectional tunnelling. The high delay demonstrates the transmission overhead in the tunnel from home network to foreign network of bidirectional tunnelling. Fig. 5 shows that about 90% of the generated messages are delivered.

3.2 Simulation model
In the simulation, we consider 16 local area networks (LANs) with a maximum of 90 mobile nodes. Each LAN has two mobility agents (i.e., one HA and one FA). All mobile nodes are allowed to roam in the network at random. The residency time for each mobile node to stay at a network (home or foreign) is drawn from an exponential distribution with a mean of $\tau$ time units. The travel time for going between subnets is exponentially distributed with a mean of $(\tau/0.9) \times 0.1$ time units. Thus, mobile nodes spend 10% of their time in transition, and 90% of their time connected to a LAN. In addition, each mobile node has a probability $p$ of losing the connection with a local mobility agent.
messages were delivered to the mobile nodes by MMP and about 50% of the generated messages were delivered by the bidirectional tunnelling protocol. For MMP, two situations may affect the delivery of multicast messages to the mobile nodes: first, the node may be in transit state; and secondly, the node may be attached to a network with poor link connection due to the noise environments. The unsuccessful deliveries in bidirectional tunnelling may be caused by inconsistent information in home network about the location of its mobile nodes.

4 Conclusions

MMP extends the mobile IP with anycast address group technology for agent discovery, registration of mobile nodes and delivery of multicast packets. The utilisation of an anycast address for the mobility agent group can reduce the cost and delay when the mobile nodes register with mobility agents between subnets without impacting its performance. In contrast to bidirectional tunnelling and remote subscriptions, MMP is more efficient in terms of delivery delay and throughput of multicast packets. The cost of employing the anycast address/group is that the multicast routers involved in the group have to manage the anycast addresses. This management may be taken as a setup cost and will not compromise the (run time) dynamic performance of MMP. In this sense, MMP will extend the performance of mobile IP, especially when multicast services are desired.

5 Acknowledgments

This work was partially sponsored by the City University of Hong Kong under grants 7001060, and UGC (University Grant Council) Hong Kong under grants 9040352 CityU 1059/98E and 9040511 CityU 1076/99E. The views and conclusions contained herein are those of the authors and should not be interpreted as necessarily representing the official policies or endorsements, either express or implied, of the UGC Hong Kong or the City University of Hong Kong.

6 References

1 PERKINS, C.E.: "Mobile IP: design principles and practices" (Addison Wesley, 1998)
2 DEERING, S.: 'Host extensions for IP multicasting'. IETF Network Working Group Request for comment 1112, 1999
4 PARTRIDGE, C., MENDEZ, T., and MILLIKEN, W.: 'Host anycasting service'. IETF Network Working Group Request for comment 1346, 1992
5 CHIKARMANE, V., WILLIAMSON, G.L., MACKRELL, W.L., and BUNT, R.B.: 'Multicast support for mobile hosts using mobile IP: design issues and proposed architecture'. Department of Computer Science, University of Saskatchewan, Saskatoon, 1997
6 BROWN, K., and SINGH, S.: 'ReMi: Reliable multicast for mobile networks'. Technical Report, Department of Computer Science, University of South Carolina, Columbia, 1996
8 JOHNSON, D., and DEERING, S.: 'Reserved IPv6 Sunnet anycast address'. IETF Network Working Group, Request for comment 2526, 1999
14 ACHARYA, A., and BAKER, A.: 'Bidirectional multicast extensions for mobile internetworking'. Department of Computer Science, Rutgers University, 1996
15 DEERING, S.: 'ICMP router discovery messages'. IETF Request for comment 1321, 1991
A Hard Real-Time Communication Control Protocol Based on the Ethernet

WANG Zhi-Ping¹, XIONG Guang-Ze¹, LUO Jin², LAI Ming-Zhi¹, and Wanlei ZHOU²

¹ Computer Science and Engineering College, University of Electronic Science and Technology, Chengdu 610054, The P.R. China
² School of Computing and Mathematics, Deakin University, Clayton, VIC 3168, Australia
wanlei@deakin.edu.au

Abstract. In this paper, we present an Ethernet-based protocol called RTCC that provides a good basis for distributed hard real-time applications without requiring any modifications to existing Ethernet hardware. Two novel mechanisms, the command/response multiplex transmission and a bus table, are introduced in RTCC in order to schedule the channels. Performance measurements from experiments on a 10 Mbps Ethernet indicate that RTCC has a satisfactory determinism.

1 Introduction

A real-time system is one in which the correctness of the system depends not only on the logical results, but also on the time at which the results are produced. In hard real-time systems, results must be produced within an ordained timing constraint, otherwise, the results will lose their usability. For safety-critical systems, incorrect operations can lead to the loss of life or other catastrophes. With the increasing use of distributed hard real-time systems (such as command and control systems, image processing and transmission, and industrial process control, etc.), the ability of computer networks to handle hard real-time message traffic is becoming more and more important. Two requirements, high bandwidth and determinism, are critical to any hard real-time applications [8]. Many existing networks, however, have either low bandwidth, or are non-deterministic, and therefore do not meet the requirements of the development of distributed hard real-time systems [1, 2, 6].

Ethernet is the most popularly used network nowadays. It is fast (10Mbps, 100Mbps or 1000Mbps Ethernet), simple and widely available. Ethernet meets the IEEE 802.3 standard, in which the channel access is random, since it is for a CSMA/CD LAN. This property leads to the unpredictable timing when sending and receiving data on the network. Therefore Ethernet is non-deterministic, which is often inappropriate for real-time work. However, the advantages of Ethernet (popularity, high-speed, and simplicity) justify the need to make it
suitable for real-time applications. In this paper, we add a layer of Real-Time Communication Control (RTCC) protocol on top of Ethernet. The RTCC protocol meets the requirements of hard real-time communication well. It also has a good real-time performance and meets the requirements of reliability for hard real-time systems. By using the RTCC protocol, an upper layer protocol can provide high-speed, reliable and hard real-time services to applications without any modification to the original Ethernet hardware.

The rest of the paper is organized as follows. In Section 2, we introduce the architecture of our RTCC protocol. The detailed components of the RTCC protocol are described in Section 3. Section 4 presents the result of performance measurement and analysis. Section 5 concludes the paper.

2 RTCC Protocol Architecture

For traditional networks, maximizing throughput or minimizing the average message delay are the most important performance criteria. In the hard real-time domain, however, the main focus is on satisfying the timing constraints of individual messages. In general, time constraints of messages are period, deadline, and so on.

To meet the time constraints, hard real-time messages must be properly scheduled for transmission. For the purpose of understanding its real-time performance, it is convenient to consider an MAC protocol as consisting of two processes: an access arbitration process and a transmission control process [5]. The access arbitration process determines when a node can send a message over the channel. The transmission control process determines how long a node can continue to send messages over the channel.

Ideally, any protocol for hard real-time communication should combine both an appropriate access arbitration process and an appropriate transmission control process. Many researchers, however, have tended to concentrate more on one process than the other. For example, the work on synchronous message transmission with the IEEE 802.5 protocol has emphasized the access arbitration process. The work on synchronous bandwidth allocation for the Timed Token protocol [6], however, has emphasized the transmission control process.

By comparing Ethernet and other network communication protocols, we have found that Ethernet nodes ask communication media positively, while it is the general case that real-time networks can only send data after the usage right of communication media is passively granted (e.g., a token or a command is received), so that conflict of network access will never happen. Hence, neither Token-Ring [1] nor Timed Token protocol [6] enjoys ideal real-time performance.

We propose an Ethernet-based communication protocol called Real-Time Communication Control (RTCC) protocol that provides a good basis for distributed hard real-time applications without the need to modify the existing Ethernet hardware. The RTCC protocol sits on top of the existing Ethernet MAC protocol, providing high speed and hard real-time communication network services to the upper application layer. The main advantage of the RTCC
protocol is that it transforms the popular Ethernet MAC protocol into a protocol that meets the requirements for hard real-time applications. Through the use of the RTCC protocol, applications can achieve hard real-time performance services without requiring any modification of existing Ethernet hardware. Figure 1 depicts this architecture.

![Network Model Diagram](image)

**Fig. 1.** The network model

The RTCC protocol adopts the main-sub command/response multiple access transmission mode. Nodes are categorized into two types in RTCC protocol. One is Bus Controller (BC), the other is Remote Terminal (RT). There is only one BC, and the rest are all RTs. The startup of message transmission and management of Bus are tasks of BC. The length and the receiver of the data will be sent by the RT immediately. Obviously the access arbitration process and transmission control process are all accomplished by BC in RTCC. By integrating the two processes and ensuring they work in phases, not only the sending time of a node's data can be assured, but also the bus time that is used by a node will be controlled.

Five components are defined in RTCC. They are frames, messages, bus table, error control and flow control, and network services for the upper layer. Their details are described in the next section.

3 The RTCC Protocol Components

3.1 Frames

Three types of frames are defined in RTCC. They are command frames, data frames and response frames (Figure 2). Command frames can only be sent by the BC to an RT, requesting the relevant RT to execute an operation. Command frames can be further categorized into data-send command frames and mode command frames. Both BC and RT can send data frames. Response frames can only be sent by an RT to the BC, as the response to a command frame from the BC, indicating current states of an RT.
3.2 Messages

RTCC refers messages as a sequence of transmission that includes command frame, response frame, data frame and state response intervals. A message is the basic unit in the transmission of the network. The transmission process of a message indicates a whole process of data transmission. There are seven classes of messages defined in RTCC. They are classified into two types: point-to-point messages and broadcast messages. Based on command frames, they can be categorized into data transmission messages and mode command messages as well. A message is called a mode command message when the command frame is a mode command frame, or data transmission message when the command frame is a data-send command frame. There are five data transmission messages in all:

data transmission: BC → RT, RT → BC, RT → RT, and
data broadcast: BC → RTs, RT → RTs.

There are two mode command messages, which are used for the management of the network. In the process of message transmission, if the message is point-to-point, once a data or command frame is received, an RT will send the response frame to BC at the first time. However if the message is broadcast, RTs do not send state frames, but the BC will poll individual RTs using the mode command. The above mechanisms are used to guarantee good predictability and high reliability, critical to a hard real-time protocol. State messages sent back from RT enable the BC to find errors or exceptions of RTs, so corresponding retrievals can be admitted and the reliability of the whole system is assured. Figure 3 shows the message formats of the RTCC protocol. There are periodic and aperiodic messages. In distributed hard real-time systems, periodic messages
are used for the transmission of periodic data streams or communication between periodic tasks of different nodes, while aperiodic messages transmit burst data or realize communication between aperiodic tasks. They are defined below:

**Definition 1.** $S_i(T_i, d_i, D_i, W_i)$ is a periodic message in RTCC protocol.
$D_i \geq T_i + d_i$, $T_i \geq d_i$, $1 \leq W_i \leq 1498$ bytes.

**Definition 2.** $A_i(d_i, D_i, W_i)$ is an aperiodic message in RTCC protocol.
$D_i \geq d_i$, $1 \leq W_i \leq 1498$ bytes.

In definition 1 and definition 2, $T_i$ represents the period of the message, namely the interval of the message stream's transmission. $d_i$ represents the time interval from the startup to the end of the message, i.e., the delay of the message transmission. $D_i$ represents the deadline of the message. $W_i$ is the length of data in the message, i.e., the data length the message is able to transmit. Excepting for the lack of certain periods, aperiodic messages are the same as period messages.

![Message Format](image)

**Fig. 3.** Message format

We give the following formulas regarding to the performance of the protocol. Utilization of messages is represented by $U$. And the utilization of a periodic message $S_i$ is:

$$U_{S_i} = d_i/T_i$$  \hspace{1cm} (1)

If the average arrival rate of an asynchronous message $A_i$ is $\lambda_i$, then the demand utilization of asynchronous messages is defined as

$$U_{A_i} = \lambda_i/d_i$$  \hspace{1cm} (2)
Besides $U$, another important performance criterion of messages is runtime overhead $L$. It is defined as

$$L_i = (d_i - W_i \times 8/Ethernet\ Bandwidth)/d_i \times 100$$

Where $W_i \times 8$ is the number of bits in a message.

### 3.3 Bus Table

The transmission of messages on the bus is performed by the execution of bus table instructions on BC. The bus table is installed on BC only, which includes a group of optimized communication instruction blocks and relevant messages. The number of source and destination nodes, the port number of nodes, the maximum delay of instruction's execution and other relevant control messages are all defined in every instruction block. Repeated appearance of an instruction is permitted in the bus table. BC executes every instruction block in the bus table sequentially. According to every execution of an instruction, there is a transmission of a message. The instructions in the bus table are circularly executed by BC, then every message gets one chance of transmission at least, and the message's real-time transmission on the bus table is assured.

Virtually the bus table is a scheduling table of the network, a key feature of real-time communication services provided by RTCC to the upper layer. The bus table can be realized in two modes: dynamic and static. A static bus table enjoys good time certainty, and can be easily realized. Its weak points are its rigid control, and modifying one instruction can result in rearranging the whole bus table, and a new schedulability test is needed. As to dynamic mode, the bus table is arranged according to current system states. Although flexible and efficient, its predictability is hard to ensure. Generally static bus tables are adopted in hard real-time communications. In the following we only discuss the static bus table.

**Definition 3.** $T(\{S_1, S_2, S_m\}, \{A_1, A_2, A_n\}, t)$ is the static bus table in RTCC protocol. It contains $m$ periodic messages and $n$ aperiodic messages, and the time length for scheduling all messages is $t$.

The arrangement of a bus table is an NP-complete problem [11]. Different approaches could be adopted according to the application's requirements, such as polling or Earliest Deadline First [3] (EDF). Generally speaking, there are the following requirements for distributed hard real-time systems to network communication:

1. to assure the period and deadline of periodic messages;
2. to assure the deadline of aperiodic messages; and
3. to be able to process soft real-time or non-real-time messages.

An algorithm for bus table arrangement is proposed here, which can ensure all the above requirements. The concrete steps of the algorithm are:
1. \( \forall A_i \in T \), if it is a hard real-time message, the Periodic Server [5] is used to assure its predictability, say, to change it into periodic message \( S_j, T_j \leq D_j - d_j \);
2. length of bus table \( l \leq \text{MAX}(T_i) \);
3. \( \forall S_i \in T \), the interval in the bus table is the period of the message \( T_i \);
4. assuring the transmission of hard real-time periodic messages first, use EDF scheduling to transmit other messages in the idle time of bus table.

We use the above algorithm to schedule the bus table: The following assertion concerns the utilization of the communication network.

**Assertion.** If the bus table \( T \) could be scheduled, then the utilization of network \( U \) should satisfy the following inequation:

\[
U = U_S + U_A \leq 1, \quad U_S = \sum_{i=1}^{m} \frac{d_i}{T_i}, \quad U_A = \sum_{j=1}^{n} \lambda * d.
\]

Parameters \( W \) and \( T \) of a message affect the arrangement a bit more. In general, the smaller \( W \) is, the easier the arrangement of the bus table becomes. By adopting the pinwheel scheduling of Sr's periodic transfer technology to transfer the period of all the messages into harmonic numbers [4], it will greatly enhance the schedulability.

### 3.4 Error Control and Flow Control

Error control and re-transmission are realized by the response frame's receipt by the BC. By analyzing the executing process of instructions in the bus table, we have the following conclusion. Under the most complex situation, the BC should receive the response frame twice to assure the transmission of data is correct. So the maximum delay of every instruction set in the bus table (i.e., the time interval of every instruction's execution) is divided into two sections \( (t_1 + t_2) \) or three sections \( (t_1 + t_2 + t_3) \), here \( t_1 \) and \( t_2 \) are the time limit of waiting, and \( t_3 \) is the time left. The BC sends every frame and sets the clock at the same time. If the response frame is not received in \( t_1 \) or \( t_2 \), or incorrect response is received, or the response frame indicates that the receiver received the frame incorrectly, the BC transmits once more, or BC requests the RT to transmit once more, or returns a failure message to assure the correct execution of the next instruction. If the time of re-transmission is less than \( t_3 \), the BC will startup re-transmission, otherwise the BC returns the error message. Re-transmission can reduce the code error ratio in the MAC layer [7]. RTCC presumes that the underlying network is reliable, so the time of re-transmission should not be too long, otherwise predictability is affected. Timely error information is also necessary in hard real-time systems.

Flow control approaches in traditional networks, such as the stop-wait protocol and the sliding window protocol are not applicable in real-time networks, because they cannot assure the predictability of network data transmission. The
RTCC protocol adopts the non-feedback flow control approach [10] for real-time communication. Through precisely setting of sending and receiving speeds, this approach enables a receiver to take old data before new data is arrived all the time.

### 3.5 Network Services for Upper Layer

The bus table builds real-time channels between nodes, from a sending port of one node to a receiving port of another node. One node can connect to more than one sending and receiving real-time channel. A real-time channel is equivalent to a message defined in RTCC. Each channel has its own time constraints. It is independent of other real-time channels, similar to an exclusive physical circuit.

RTCC provides two basic functions for upper layer accessing real-time channels, \texttt{rtccSend} and \texttt{rtccRecv}. Upper layer software can call the two functions to transmit data directly or extend the network functions based on them, such as transmitting data of more than 1498 bytes, providing synchronous and asynchronous communication functions, etc. The two functions are defined as follows:

- \texttt{rtccSend(sendPortNum, pBuffer, size, overwriteFlag)}: Send data to a sending port \texttt{sendPortNum}. Where \texttt{pBuffer} is the address point of sending data; \texttt{size} is equal or less than the length of data that the real-time channel can send at one time; and overwriteFlag stands for the written flag. If overwriteFlag is \texttt{TRUE}, the new data can overwrite the data that has not been delivered. If it is \texttt{FALSE}, the new data can not overwrite the undelivered data, and the error return value indicates that the old data is not sent yet.

- \texttt{rtccRecv(recvPortNum, pBuffer, size)}: Get the data from the receiving port. Where \texttt{pBuffer} is the address point of receiving buffer; \texttt{size} is the length of received data.

### 4 Performance Measurement and Analysis

The experimental setup of the RTCC protocol contains three Intel EV896EX embedded computers (clock rate is 25MHz, PC/104 bus). One is assigned as the BC, and the other two are assigned as RT. Each computer has a UM9008 type 10M Ethernet card that connects to a DE109-TC type HUB. The averages of message delays were measured through experiments. The delay of point-to-point mode command messages is 103\textmu s. And the delay of broadcast mode command messages is 52\textmu s. All of the values measured have a deviation of \pm 9\textmu s from the average values. The variation has no distinct relation with the difference of data size of messages. The results show that the real-time channels have good real-time features. To the best of our knowledge, other implementations of real-time communication protocols on Ethernet are generally soft real-time (such as RETHER [9]). Therefore it is difficult to compare them with RTCC. Figure 4(a) shows the comparison of RTCC and the hard real-time communication protocol MIL-STD-1553B in terms of delays of messages as the data size of
messages increasing. Figure 4(b) displays the runtime overhead of messages of these two protocols. Table 1 lists some main performance criteria of several types of commonly used bus real-time networks. It can be found from table 1 that RT-Ethernet (an Ethernet using the RTCC protocol) has a good real-time performance, a higher bandwidth, and a larger data size a message is able to send at one time. So it should have a better potential to be widely adopted.

<table>
<thead>
<tr>
<th>Table 1. Features of Real-time Networks</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Speed (bps)</strong></td>
</tr>
<tr>
<td>-----------------</td>
</tr>
<tr>
<td>Determinism</td>
</tr>
<tr>
<td>Length (km)</td>
</tr>
<tr>
<td>Max nodes</td>
</tr>
<tr>
<td>I Frame</td>
</tr>
<tr>
<td>Applications</td>
</tr>
</tbody>
</table>

Fig. 4. RTCC vs. 1553B: (a) Delay of Messages; (b) Time Overhead

5 Conclusions

The Ethernet-based RTCC protocol proposed in this paper uses command / response multiplex transmission and a bus table to schedule real-time messages. Through analysis and experiments we have shown that RTCC has a good real-time performance, is a simple protocol, and meets the requirements of real-time communications well. RTCC also meets the requirement of reliability for hard real-time systems. RTCC has been used in a national essential project of China in the aviation area and a monitoring system for patients in danger successfully.
Further work has been planned. That includes (1) Extending RTCC protocol to support multi-segment Ethernet (connected through bridges or routers), and (2) Studying the possibility of applying the RTCC in distributed soft real-time systems (such as distributed multimedia systems).

References

AN OBJECT-ORIENTED DESIGN PATTERN FOR DISTRIBUTED REPLICATION SYSTEMS

LI WANG and WANLEI ZHOU
School of Computing and Mathematics
Deakin University, Geelong
Australia, Vic 3217
E-mail: {liwang, wanlei}@deakin.edu.au

Abstract

Replication is a key to providing good performance, high availability, and fault tolerance in distributed systems. However, designing and implementing a replication system is a very difficult task. Based on an active replication model, this paper focuses on an object-oriented design pattern to simplify the implementation of distributed replications. The pattern addresses the following design approaches: (1) A replica is an integration of two kinds of objects, service implementation objects (service object in short) and a replication management object (named as surrogate object). The service object implements a set of services exported to clients. The surrogate object implements a replication policy. (2) Clients are bound to a replicated service through a naming service. The naming service is updated by replica groups so that dynamic membership changes are passed through the naming service to clients. (3) Group communication services, providing support to the replica group, are embedded in the surrogate object.

Keywords: distributed service replication, distributed object, fault-tolerance, group communication.

1 INTRODUCTION

Distributed replication is the maintenance of copies of software and/or data on different sites. It is a common means by which a service system can provide continuous services in the presence of failures. Replication schemes generally follow two streams, namely primary-backup replication and active replication [9, 6]. In the primary-backup approach, replicas are designated as the primary and backups. Only the primary interacts with clients (processing requests and sending replies), backups stand by and wait for the state changes from the primary at certain checkpoints. In the active replication approach, replicas are not distinguished from each other. All replicas interact with clients. Active replication is also referred as state machine approach [10]. A state machine is used to model each replica's a deterministic process. The consistency between replicas is preserved by delivering the same sequence of ordered operations to each replica.

This paper deals with the active replication schemes. We propose an active model where data consistency is defined by all operational replicas delivering the same set of update requests. A distributed replication system well falls into the distributed process group concept. Through the I3 project [3] and other similar projects [8], group communication services as the basis for supporting process groups have emerged to matured techniques. Group communication services provide reliable and ordered multicasting mechanisms that simplify programming process groups substantially. This paper gives in detail about two of group services, atomic multicasting and dynamic membership services.

Design a distributed replication system is a difficult task. This paper is aimed to promote distributed object methodologies to bring flexibility and modularity into distributed replication systems. Java is used in this paper as the target language for implementing the prototyped replication system. Java, as a fully object-oriented, platform-independent, and Internet-friendly language, has gained enormous recognition [5]. Through the experience of authors, it is indeed proven a neat and easy-to-work-with high-level language for developing network programs. Java-like pseudo codes, especially public interfaces, are used throughout this paper to demonstrate various design ideas.

The rest of the paper is organized as follows. Section 2 describes the object-oriented system model. Section 3 presents the group services to facilitate the development of the active replication scheme. Section 4 gives the implementation overview. Section 5 summarizes the paper.

2 THE SYSTEM MODEL

We model a distributed replication system as a group of replicas (or members, term replica and member are used interchangeably in this paper) located on a set
of sites, only one replica at a site. The functionality of a replica is two-fold, providing a set of services and implementing the replication policy. The replication system as a whole should appear as a single service provider to clients. This section presents the active replication scheme, the internal structure of a replica, and the naming service.

2.1 The Active Replication Scheme

Figure 1: THE ACTIVE REPLICATION MODEL

Figure 1 depicts our active replication scheme. We list the following characteristics of the proposed active scheme.

1. A client \((c_i)\) is connected to only one member \((R_j)\) of the group at a time. This member is responsible for processing and replying the client’s requests.

2. An update request received at a member is multicast to all other members before being delivered. A query request is delivered and processed locally.

3. Data consistency is defined by all operational members delivering the same set of updates in FIFO order. Requests ordering, such as causal ordering and total ordering \([1]\), are not considered in the paper.

In this active replication scheme, loads can be shared by group members. The response time of a query is shortened as clients are not lining up at one queue, but at different queues. However, an update request may imply a longer response time as it has to be propagated to full members of the group before being delivered.

2.2 Distributed Object Construction

Basically, an object is defined by some state variables (holding the data) and a set of encapsulated operations (processing the data). The state variables can only be accessed exclusively through the set of operations. A replica is designed as an integration of two kinds of objects, service objects and the surrogate object, to separate the services provided and the replication policy adopted.

The service object implements a set of services which are exported to users. The service object is purely interested in providing correct services without being involved in matters like where it should be located or whether it will be replicated. It is a distributed object that can be freely moved onto any site. The granularity of service objects could be from as large as a database manager, to as small as a queue manager. Services provided are classified as either queries or updates. A query does not change the state of the service object but an update does.

A set of replicated surrogates form as an object group that implements the replication policy. The following Java skeleton codes shows the service object\(Service\), the Surrogate object, and the integration interface between them through dispatch(). 

\[
\text{dispatch()} \text{ is an abstract method of the surrogate object, it is implemented according to what service objects are integrated with the surrogate. The dispatch()} \text{ issues up-calls to relevant methods of service objects in accordance with clients' requests.}
\]

\[
\begin{align*}
\text{public interface Service} & \{ \\
\text{public service.1();} \\
\text{...} \\
\text{public service.n();} \\
\} \\
\text{class ServiceImpl implements Service} & \{ \\
\text{abstract class Surrogate} & \{ \\
\text{private ServiceImpl service;} \\
\text{public Surrogate (ServiceImpl service) \{ \\
\text{public abstract void dispatch (\ldots);} \} \\
\text{public static void main O \{ \\
\text{ServiceImpl service = new ServiceImpl();} \\
\text{Surrogate replica = new Surrogate (service);} \\
\text{replica.dispatch(\ldots);} \} \\
\} \\
\}
\end{align*}
\]

2.3 The Naming Server

The replica group is designed to be dynamic in the sense that they do not operate on a static set of sites, rather, any site can be used. The group membership of the replication system is changed under three kinds of events: join/leave/crash. Section 3.3 will discuss in detail what properties have to be preserved during membership changes.

A naming server\(NS\) is established to sit between clients and the group. We assume the NS is installed at a stable site that does not crash. By doing so, the replication group is isolated from clients. Clients do not need to keep the membership of the group, rather only have to know the address of the NS and the symbolic name of the replication group. By sending a request to the NS asking for a binding to a replicated server, a client is allocated with a member of the group. Then the client makes the direct connection to that member.
which becomes the accessing point of the replicated service. When that member becomes unavailable, the client sends another request to the NS for the next available member. Membership changes of the NS are updated by the replication system. So the dynamic membership is passed to clients in a transparent way.

The NS can employ some load balancing policy to assign group members to clients so that loads of group members are roughly even. We form a ring of operational members in the naming server, a new client is bound to the next member in the ring. Thus the clients are evenly allocated to members of the group. This interface can be used to manage multiple replication groups. Here we give the NS interface:

```java
public interface NamingService {
    void createGroup(String[] members, String groupname);
    void addMember(String groupname, String member);
    void deleteMember(String groupname, String member);
    String bind(String groupname);
}
```

class NSImpl implements NamingService {
}

3 GROUP COMMUNICATION

Group communication studies the services that provide support to a group of distributed processes in their coordinations and synchronizations. Group services fall into two major categories, reliable and ordered multicasting services, and the membership service. In this paper, we focused on two of them, the atomic multicasting service and the dynamic membership service. The multicasting atomicity service guarantees members receiving the same history of updates. Whereas the dynamic membership service guarantees that membership changes do not bring inconsistency to the replication system. With the support of group services, the complexity of designing replication systems is simplified substantially.

3.1 Assumptions

Without considering failures, the implementation of group services is trivial. However, group services need to be fault-tolerant to provide expected services in the presence of failures. Here we give the failure semantics, failure detection methods, and the basic communication assumption that the multicasting atomicity and the dynamic membership services are based on.

Failure semantics. Any group member may fail and then becomes unavailable. A failure can be caused by following events:

1. Voluntary-exit. A member exits voluntarily due to, such as lacking of resources, or being switched off by the system administrator. In this case it can restart at a well defined state.

2. Process crash. A member process stops the execution due to various reasons, such as being killed deliberately. It may not be able to restart at a well defined state.

3. Site crash. The host a member is running on has crashed. The member definitely cannot restart at a well defined state.

Failure detection. Each member is monitored by other members of the group. In accordance with above three failure semantics, we use three methods for detecting the failures:

1. Voluntary-exit. The member is about to exit voluntarily sends a notification to all other members of the group prior to the exit.

2. Process crash. A hint from the operating system can be used to detect the failure. The failure may appear in the form of an exception raised by the operation system.

3. Site crash. Timeouts is used to detect a failed member in a synchronous environment.

Using timeout for detecting failures relies on whether the underlying distributed environment is assumed to be synchronous or asynchronous. Synchronous environment assumes a known upper bound on message transmission delays. Asynchronous environment does not impose this upper bound, thus a site being very slow can not be distinguished from actually have failed. Asynchronous assumption is more realistic but hard to implement. We assume the synchronous environment in this paper. For more comprehensive understanding of what properties hold for synchronous and asynchronous group communication services, refer to Crasian [4].

Reliable FIFO communications. Reliable and FIFO communication is assumed between a sender and a receiver. This implies messages sent from a sender to a receiver do not get lost and arrive in FIFO order at the receiver. This reliable FIFO communication assumption can be easily mapped to the stream-based (TCP/IP) network connection provided by general operating systems.

3.2 Atomic Multicasting

A multicasting service is a service of sending a message to multiple receivers. Multicasting atomicity here is defined as receiving atomicity, i.e. a message is either delivered by all operational receivers or by none of them. In other words, multicasting atomicity exhibits
all-or-none property. In [7], it used atomic broadcast to include both atomicity and total ordering of messages delivered by all operational receivers. We only consider the atomicity in this paper. The multicasting atomicity is used in the active replication scheme by members propagating update requests to other members, and also in maintaining group membership changes. In fact, it is used as the basic communication primitive for all communications between members.

When a sender multicasting to a set of receivers, the message is sent multiple times using point-to-point connections to receivers. If the sender does not crash during the multicasting, the atomicity is easily guaranteed. However, if the sender crashes in the middle of the multicasting, the crash may lead to inconsistency where some members get the message, some do not. Based on the assumption of FIFO reliable communication between senders and receivers, without loss of generality, we assume a member making atomic multicasts in the sequential order, i.e. the member does not start an atomic multicast before finishing the previous one. Thus, when a member receives a multicast from the same sender, it implies the previous multicast should have completed. In turn, when the sender crashes, only the latest multicast sent might not reach every receivers.

Multicasting atomicity protocol. That a multicast is not confirmed to have reached to all receivers is referred to the unstable multicast [2, 10]. To achieving multicasting atomicity, those members who have received the unstable multicast need to re-multicast it upon knowing the sender's failure. The basic idea of the protocol is to let operational members exchange (re-multicast) their last received message (the unstable multicast) from the failed member.

3.3 Dynamic Membership Service

Membership is the information about a group that shows who is in the group. The membership is changed by a member's join/leave/crash event. In essence, the membership is replicated at each member and also the NS, so a multicast can be sent to those who are currently operational, and also clients can connect to new members via the NS.

The membership is represented by a view consisting of members of \( \{g_1, \ldots, g_n\} \). Members start with the same local view, \( \text{view}_0 \). Let \( \text{view}_i, i = 0, 1, \ldots \) be the successive views of the group, \( \text{view}_i \) and \( \text{view}_{i+1} \) are different only by an addition or a deletion of one member. A view update message \( m_{\text{join}}/m_{\text{leave}}/m_{\text{crash}} \) is multicasted to members in \( \text{view}_i \cap \text{view}_{i+1} \), changing \( \text{view}_i \) to \( \text{view}_{i+1} \).

Total ordering of membership changes. View changes are delivered in total-order at all members. Total ordering is accomplished by designating a member as the coordinator. A join/leave/crash request is sent to the coordinator from where it is multicasted to all other members. If the coordinator crashes, the new coordinator is selected by an algorithm, e.g. the oldest member in the group.

![Figure 2: The Member Joining Scenario](image)

Member joining. A joining event is a concurrent event with multicasts among group members, a scenario is depicted by Figure 2. \( g_1 \) is the new member joining the group. At the same time, a multicast \( m_i \) by \( g_2 \) is sent to current view \( \{g_1, g_2, g_3\} \), which is delivered at \( g_2 \) and \( g_3 \) before \( m_{\text{join}} \), but after \( m_{\text{join}} \) at \( g_1 \). Suppose \( g_1 \) is the coordinator, it transfers its state to the new member at the instant of receiving \( m_{\text{join}} \). Later on, when \( g_1 \) delivers \( m_i \), \( g_1 \) will miss it.

Virtual synchrony on membership service. The virtual synchrony model [2, 1] developed by Isis project [3] defines the membership atomicity, which is a property that messages sent in \( \text{view}_i \) have to be delivered in the same view. That is, a \( m_{\text{join}}/m_{\text{leave}}/m_{\text{crash}} \) is delivered at all members of \( \text{view}_i \cap \text{view}_{i+1} \), when messages sent in \( \text{view}_i \) are group-wide stable. Membership atomicity is implemented by the flush protocol [2].

Member leaving. Conceptually, the flush protocol is not needed when a leaving event \( m_{\text{leave}}/m_{\text{crash}} \) happens, since state transfer is not involved. But flush protocol can serve as a checkpointing mechanism that a member leaving voluntarily can have a consistent state with the group at the instant of departure. Later, this member can rejoin the group by catching up missed operations. The catch-up might be a more appropriate solution than state-transfer in the situation where the group manages a large volume of data.

When dealing with a member leaving by crash, both the multicasting atomicity protocol and the flush protocol have to run. The multicasting atomicity protocol stabilizes the last multicast of the crashed member group globally, whereas the flush protocol guarantees messages are delivered in the same view at all members. Therefore, these two protocols are combined into one protocol as the multicasting and membership atomicity protocol in the event of a member's crash.

Membership consistency between the group and the NS. A member designated as the coordinator is responsi-
ble for updating the NS right after its local view is updated. Hence, the NS executes the same set of membership updates in the same order as the group.

4 SYSTEM IMPLEMENTATION

There are three kinds of components in the system, replicas, clients, and the naming server. Figure 3 depicts the relationship of these components. A replica is composed of two objects, the service object and the surrogate object. We have presented the skeleton codes for the service object and the surrogate object in Section 2. This section will focus on the internal design of the surrogate object, which is a coherent integration of many other objects. Objects are layered so that the implementation of the surrogate is made easily. Figure 4 presents the layers of objects supporting the surrogate.

Service Objects
Surrogate Object
Mini-Protocol Objects
Utility Objects
OS/Networks

Figure 4: THE OBJECT LEVELS

Utility Objects. Utility objects manage basic resources such as local group view, messages, communication channel, etc.

Group. A Group object manages the local view of the operational members. It basically provides two methods, leaveGroup(), and joinGroup(), to remove a member and to add a member respectively.

A member is fully connected with all other members. The replicated server group can be created in a number of ways: (1) The group is created by taking a list of members as the input. (2) The group is created with empty members, then each member joins the group.

CommToMembers. A CommToMembers object sets up the stream-based connections to each members in the group. Streams are collected into a vector, channels, which is used for multicasting.

Threads. Multi-threading gives rise to efficiency. Each surrogate runs multiple threads to handle different matters so that the main thread is not blocked. There are three types of threads: AcceptClients, AcceptClient, and ReceiveMcast. AcceptClients is the parent thread for accepting all clients' connections, and it spawns a child thread (AcceptClient) to each connection with a client. A ReceiveMcast is a thread to read all multicasts from peer members so that the main thread can be free from this matter to concentrate on request handling.

Message. A message object is used to form messages used in the system, it attaches a sequence number to each multicasted message. Messages are classified in order to direct different actions. They are part of protocols. Messages are formed by using a three-tuple format:

| Message := (type, mid, message-content); |
| Type := 'u'/'p'/'v'/'f'/'h'/'r'; |
| mid := sender-address:message-no; |
| ('u', mid, update-request). Client update requests. |
| ('p', mid, query-request). Client query requests. |
| ('v', mid, propagated-update-request). Propagated update requests to the members of the group by the member first receives it. |
| ('h', mid, joining-member). Representing a joining event, a m_f is sent by the joining member to the coordinator. m_f triggers the flush protocol. |
| ('f', mid, leaving-member). Representing a voluntary leaving event, a m_f is sent by the leaving member to the coordinator. m_f triggers the flush protocol. |
| ('h', mid, failed-member). Representing a crash event. Upon detecting a crashed member, the member sends m_f to the the coordinator. m_f triggers the combined protocol of the multicasting and membership atomicity. |
| ('f', mid, status-of-the-member). m_h is used by the flush protocol. |
| ('r', mid, status-of-failed-member). m_r is used by the combined protocol. |

Mini-protocols. The system uses objects to accomplish the execution of protocols. Currently we have two mini-protocol objects: Flush and McastAtomAndFlush. The Flush object is used to perform the flush protocol, whereas the McastAtomAndFlush object performs both the multicasting atomicity and the flush protocols.

The construction of the surrogate. The surrogate is the replica which runs on a site. The surrogate object instantiates and integrates with utility objects and mini-protocol objects, and controls the flow of executions. The state variables of the surrogate includes: \{G, InQ, OutQ, last_mcast[G]\}. G represents the local view of the group membership. InQ stores all incoming client requests and messages from peer members of the group. Requests stored in InQ are delivered in FIFO ordering. OutQ stores outgoing results. last_mcast[G] is used by the atomic multicasting protocol.

Current status of the research. Experiments regarding to performance studies are being conducted under Java JDK 1.1.6 version. We also intend to de-
velop various replication policies and form Java packages so that fault-tolerance can be achieved at different levels of strength.

5 REMARKS

A design pattern for distributed object replications has been presented. The pattern promotes the following design ideas:

- Service implementations are separated from replication policies. The service implementation is encapsulated by service objects, whereas the replication policy is implemented by the surrogate object. The surrogate object is the replica who runs on a site. The surrogate instantiates utility objects and mini-protocol objects, and controls the flow of executions. In reality, the surrogate propagates update requests to other surrogates, issues up-calls to the service objects, sends replies to clients, and executes protocols in the event of a member’s join/leave/crash.

- Two group communication services, multicasting atomicity and membership atomicity, are implemented as mini-protocol objects. These two group communication services guarantee that members receive the same history of updates and also that membership changes do not bring inconsistency to the replication system.

- A naming service is introduced to manage the dynamic membership changes and to pass these changes to clients transparently.

This design pattern is an extension of stub design ideas innovated by remote procedure call (RPC) systems. Stubs are used to hide network connection and parameter marshalling details from programmers, whereas the surrogate designed in this paper extends stubs with replication mechanisms. The pattern is general enough to the design of any object replication system.

References


Primary-Backup Object Replications in Java

Li Wang and Wanlei Zhou
School of Computing and Mathematics
Deakin University, Waurn Pond, Vic 3217
Australia
E-mail: {liwang, wanlei}@deakin.edu.au

Abstract

Service replication is a key to providing high availability, fault tolerance, and good performance in distributed systems. Various replication schemes have been proposed, they are based on two streams of techniques, namely passive replication and active replication. This paper focuses on two implementation approaches of the passive primary-backup scheme, remote method invocation approach and replica-proxy approach, using Java RMI and Java network packages respectively. Issues addressed in this paper also include: the primary-backup protocol, restarting a failed server at any site; and a general naming service for the maintenance of dynamic memberships of replica groups. Finally, performance studies based on two implementation approaches are given.

Keywords: Service Replication, Distributed Object, Fault-Tolerance, Remote Method Invocation

1: Introduction

Distributed replication is the maintenance of copies of software and/or data on different sites. It is a common means by which a distributed system can provide continuous services in the presence of failures. Replication schemes generally follow two streams, primary-backup and active replication [3, 5, 6]. In the primary-backup approach, replicas are designated as the primary and backups. Only the primary interacts with clients (processing requests and sending replies), whereas backups stand by and wait for state changes (or updates) from the primary at certain checkpoints. According to how frequently the checkpointing is taken place, primary-backup replications are further categorized into the hot scheme (every state change is propagated to backups right away), the warm scheme (a set of state changes are propagated at checkpoints), and the cold scheme (no propagation) [1]. Primary-backup schemes are called passive replication for the reason that backups passively receive state changes without having any interaction with clients.

In the active replication, all replicas interact with clients. A query request is handled by one replica, an update request is dispersed to all replicas. In contrast to the passive replication, it is named active for the reason that all replicas are actively taking clients’ requests. Active replication is also referred as the state machine approach [7]. A state machine is used to model each replica as a deterministic process. The consistency between replicas is preserved by delivering the same sequence of ordered operations to each replica.

Most of the research on primary-backup replication [3, 5] is concentrated on consistency protocols of employing multiple backups. It is argued though, if we can restart a failed replica (primary or backup), there is no need to employ a collection of backups, provided the primary and the backup do not fail at the same time. Thus, we demonstrate one backup approach in this paper.

The aim of this paper is on two implementations of primary and backup objects in Java, remote invocation method (RMI) approach and replica-proxy approach. The RMI approach uses the Java RMI [8] tool as a vehicle to implement primary and backup objects. It is a simpler and faster implementation approach, but less flexible. Whereas the replica-proxy approach based purely on Java network packages is a slower implementation, however, brings a better performance to final systems.

The rest of the paper is organized as follows. Section 2 presents the system model containing the primary-backup protocol, failure semantics, and a naming service. Section 3 describes two implementation approaches in detail. Section 4 studies the performance. Section 5 concludes the
2: System model

As depicted by figure 1(a), there are three types of software entities involved, namely clients, the primary, and the backup. Each entity is constructed by an integration of objects. Basically, an object is composed of some state variables and is encapsulated by a set of operations. The state variables can only be accessed exclusively by the set of operations. Operations are categorized as either queries or updates. A query does not change the state of the object but an update does.

Choosing the right checkpointing policy (hot, or warm, or cold) depends on the environment in which the replication system is running, and the requirement on the fail-over time which is the time period from the primary’s breakdown to the backup taking over. Obviously, the hot policy gives the shortest fail-over time.

2.1: The primary-backup protocol

We chose to use the hot policy so that the primary-backup system can be recovered without any delay. Figure 1(b) shows the primary-backup protocol.

- **Client side.** Clients send requests only to the primary.
- **Primary side.** The primary propagates each update request to the backup. After the backup returns the acknowledgement, the primary replies to the client.
- **Backup side.** Upon receiving a propagation from the primary, the backup executes the request and acknowledges.
- **Unique primary at any time.** Both the primary and the backup agree that there is a unique primary at any time.
- **Replica consistency.** The primary and the backup execute the same set of operations in the same order.

2.2: Failure semantics

A failure is defined by either a process failure or a site failure. The process failure implies the process has terminated by an exit or by a crash. The site failure implies the host in which a replicated server is running has crashed. The process failure can be detected by a hint from the underlying operation system, whereas the site failure can only be suspected using time-outs.

In the primary-backup replication model, the primary monitors the backup. Upon detecting the backup’s failure, the primary disconnects with the backup and will not propagate any subsequent requests. Those subsequent requests are saved into a log file and transferred to the backup when the backup is restarted. Clients monitor the primary. Upon suspecting the primary’s failure, clients switch to the backup and re-sends their last request. Duplicated requests from clients can be filtered out by attaching sequence numbers to requests. The following gives the life-cycle of both primary and backup objects:
• **The life-cycle of the primary.** It is started. → It fails. → It is restarted as the backup. → It joins the primary. → It becomes the new primary when the primary dies.

• **The life-cycle of the backup.** It is started. → It joins the primary as the backup. → It becomes the new primary when the primary fails; Or it is started. → It fails. → It is restarted as the backup.

2.3: Naming service

In practice, the failed primary or backup are restarted as soon as possible. However, when a failed one is restarted at a different site, this new site has to be passed to clients. There are generally two methods to solve this problem. First, when the backup is restarted at a new site and joins the primary, the primary piggybacks the new reference to clients. Second, a naming service (NS) is set up separately on a stable site to manage the site changes. We choose to use the NS approach to deal with site changes as it can be extended to a general service provider for maintaining multiple replication groups. The NS interface is then defined as follows:

```java
public interface NamingService {
    void createGroup(String[] members, String groupname);
    void addMember(String groupname, String member);
    void deleteMember(String groupname, String member);
    String bind(String groupname);
}

class NSImpl implements NamingService { }
```

When a replica group is created, the coordinator of the group registers the group reference with the NS (by invoking `createGroup()`). The coordinator is also responsible for updating the NS with any member changes (by invoking `addMember()` or `deleteMember()`). For the primary-backup scheme, the primary is the coordinator, which invokes `deleteMember(thegroup, backup)` upon the backup's failure, and `addMember(thegroup, backup)` upon the backup's joining. Whereas clients use `bind(group)` to get the reference to the primary.

3: Implementation approaches

This section discusses the implementation detail based on two approaches, RMI approach and Replica-proxy approach. Both primary and backup execute the same code fragments but are started differently as being either the primary or the backup. Implementation approaches follow the protocol depicted by figure 2(b).

![Figure 2. (a)RMI Approach. (b) Replica-Proxy Approach](image)

3.1: The Java RMI implementation

A RMI (the same concept as remote procedure call (RPC)) system generally provides a separate IDL language to define the interface for a remote service. Using a RMI system to implement primary and backup objects implies clients making RMI calls to the primary, and the primary making RMI calls to propagate client calls to the backup. Figure 2(a) depicts this approach.
Java RMI does not introduce a separate IDL language, rather it introduces RMI packages for the implementations of remote objects. Thus, communications among clients, the primary, and the backup are via three RMI interfaces: the service interface, the primary-backup interface, and the replica interface. We use a service ROOMBooking as an example. Two operations are provided: book() is an update operation, and booking() is a query operation. The interface between the primary and the backup has three methods: (a) join() is used by the backup to join the primary; (b) stateTransfer() is used to transfer the state of the primary to the backup at the instant of joining; (c) When the primary or the backup suspecting the failure of the other, it uses a kill() to confirm the decision. The killed one has to be restarted and then rejoins the primary. The following code fragment shows these interfaces:

```java
public interface ROOMBooking extends Remote {
    public boolean book (String booking);
    public String allBookings (Date D);
}

public interface PrimBackup extends Remote {
    void join (String backup);
    void stateTransfer();
    void kill();
}

public interface Replica extends ROOMBooking, PrimBackup {
}
```

The primary-backup system can be started in a couple of ways: (1) The primary is started first, the backup is started with the reference to the primary and joins the primary. (2) The backup is started first, the primary is started with the reference to the backup and the primary sets up the connection to the backup.

3.2: The Replica-Proxy implementation

This approach builds up the replication system based on networking packages java.net instead of the RMI system. Replica proxies are used as replica stubs that service objects can plug in and play. In turn, replicas are an integration of two kinds of objects, service objects such as ROOMBooking service, and replica-proxy objects. A service object provides services without being involved in matters like where it should be located or whether it is replicated. The replica proxies deal with network connections, message marshallings, and more importantly replication managements. Figure 2(b) depicts this approach.

The proxy object is designed by using: (1) A dedicated communication channel is set up between the primary and the backup. (2) A parent thread is used to accept all clients' connections at any time. Upon each connection, the parent thread spawns a child thread to deal with that client. (3) A thread is started by the primary to accept the backup so that the backup can join at any time.

4: Performance studies

The performance study is conducted within a local subnet (10M Ethernet) of Sun sparce stations under Java JDK 1.1.2 version. The tests use one client making synchronous (blocking) requests to the primary. The tests are performed by measuring one round-trip time of sending a request and receiving different sized replies. In figure 3, X-axis represents the reply message sizes (in Kbytes), whereas Y-axis represents the response time (in msec) of average 100 requests. Two groups of experiments are performed based on two implementation approaches. Both are based on two settings: (1) a single primary; (2) the primary and the backup. Figure 3 shows the test results:

1. With the RMI approach, depicted by lines RMI - P (single primary) and RMI - PB (primary and backup), the tests show there is some performance penalty involved by introducing the backup. This is largely because two RMI calls are issued before replies are sent back to clients.
2. With the replica-proxy approach, the tests show that introducing the backup does not bring a significant performance penalty. The two lines, Proxy-P (single primary) and Proxy-PB (primary and backup), are very close to each other and crossed sometimes. Also we can
Figure 3: The Performance of RMI and Replica-Proxy Implementations in Java

observe the replica-proxy approach is better performed than the RMI approach. This is because of the optimized codes in the replica-proxy implementation, in particular, the backup only sends acknowledgements to the primary instead of request results.

5: Remarks

In this paper we have presented the design, two implementation approaches, and the performance studies of the primary-backup replication scheme. Generally speaking, the RMI approach is a faster, but less flexible implementation approach. The replica-proxy approach is a slower process, but results in better performance and optimized codes, especially, performance can be better achieved when replies involve a large volume of data. More complex replication schemes can be better implemented using the replica-proxy approach, which can serve as a general design pattern for any object replication system, such as the active replication schemes.

We also introduced a naming service for maintaining dynamic replication groups so that a failed replica can be restarted at any site. In practice, however, the failed one is often restarted at its original site.

Java, as a fully object-oriented, platform-independent, and Internet-friendly language, has gained enormous recognition [4]. Through the experience of authors, it is indeed proven a neat and easy-to-work-with high-level language for developing fault-tolerant and distributed programs.

References

A Tool for Constructing Service Replication Systems

Li Wang and Wanlei Zhou
School of Computing and Mathematics
Deakin University
Warrnambool, Vic 3271
Australia
E-mail: {lwang, wanlei}@deakin.edu.au

Abstract

Service replication is a key to providing high availability, fault tolerance and good performance in distributed systems. However, building a service replication system is a difficult and complex task. This paper describes a tool that mimics the design of the remote procedure call (RPC) system to support building distributed service replication systems. The tool includes an interface definition language for describing a replica group, a language preprocessor and a runtime library system. Keywords: Service Replication, Distributed System, Interface Definition Language

1. Introduction

Replication in general is an important approach to increasing the availability, achieving fault-tolerance, and improving the efficiency of a system. Replication is a technique of duplicating a critical data item and/or software entity on multiple sites so that one server/site failure can be tolerated by the replicated servers (replicas). Better performance also can be achieved by replicas executing client requests concurrently.

There are some toolkits developed by various research groups, such as group communication toolkits like Lisa [1], and toolkits built on group communication toolkits like GARF [2], etc. These toolkits claimed to support building reliable distributed systems and can be used to build service replication systems. However, those toolkits mainly provide library-level support. The developing process based on these toolkits is still manual — programmers have to be very familiar with the toolkit and to implement the replication control protocol and the whole system by themselves.

The purpose of this paper is to describe a tool aimed at supporting the construction of distributed replication systems. The tool includes a language for specifying a replica group; a language preprocessor that generates relevant codes to assist forming client programs and replicated server programs; and a runtime system that supports the execution of both client and replica programs. With this tool, programmers are free of dealing with network communications and replication control protocols, these protocols are generated by the tool automatically. Therefore, programmers can concentrate on the implementation of application level functions.

The rest of the paper is organized as follows. Section 2 presents an integrative replication scheme. Section 3 describes the object-based software entities. Section 4 gives the tool components. Section 4 shows an example based on the primary-peer scheme. Section 5 concludes the paper.

2. An Integrative Replication Scheme

Replication schemes generally follow two styles, the primary-backup scheme [3, 5] and the active replication scheme [3, 4]. In the primary-backup approach, replicas are distinguished as the primary and backups — the primary interfaces with clients, backups stand by and await state changes being propagated from the primary at a time interval. However, it can be easily seen that the primary becomes the communication bottleneck since all requests are sent to it. To provide better performance, the active replication scheme was proposed [3] to let replicas execute client requests concurrently.

However, the active scheme introduces a subtle problem known as message ordering [1]. Since clients sending requests concurrently, the order of requests arrives at each replica may be different due to the network bandwidth/traffic and speeds of different sites. Fig. 1(a) depicts this scenario.
Figure 1. (a) The ordering scenario. (b) The primary-peer replication scheme.

\( R_1, R_2 \) and \( R_3 \) are three replicas. Two clients, \( c_1 \) and \( c_2 \), send requests to the replica group concurrently. Requests, \( r_1, r_3 \) and \( r_2 \), arrive at \( R_1, R_2 \) and \( R_3 \) in the orders of \( < r_1, r_3, r_2 > \), \( < r_2, r_1, r_3 > \) and \( < r_1, r_2, r_3 > \) respectively. Thus, the states of replicas may be divergent by executing the same set of operations in different orders. In turn, the data consistency is violated. In general, messages among members of a group have ordering constraints classified as: first-in-first-out (FIFO), causal total, and total-causal orders [1].

To accommodate both styles of replication schemes, we propose an integrative scheme which is based on the idea of the active replication, but it is configurable to the primary-backup scheme. The scheme is named as primary-peer replication scheme (PPRS), where one replica is distinguished as the primary, others as peers. A primary-backup group is configured simply by directing all requests to the primary of the group and leave peer members (now backup) only receiving propagations. Fig. 1(b) depicts the architecture of the PPRS. In the PPRS, each client is allowed to connect to one replica at a time, update requests received at each replica are propagated to other replicas at a specified interval.

3. The Object-Based Software Entities

According to Fig. 1(b), we can extract two basic software entities in the system: client and replica. To separate concerns, a client entity is composed of two objects: a client object and a client proxy. The client object is an application level program. It often provides a user interface (such as a graphic user interface) to access operations provided by a server. Whereas the client proxy is generated by the tool. The proxy object handles the communications with the replica group and provides the replication and distribution transparency to the client object.

By analogy, a replica (server) entity is composed of two objects as well: a service object and a surrogate object. The service object implements a set of operations exported to clients, whereas the surrogate handles network communications with client proxies and peer replicas. The replication control protocol is embedded within each surrogate. The protocol guarantees that all update requests will be propagated to full membership in a sensible order that meets the data semantics of the application, despite any failure.

The tool also provides a group management service (GMS) as a bridge between clients and the replica group so that any membership change can be passed over to the clients. A replica group is registered with the GMS under a unique group name, and the primary member of the replica group is responsible for updating the membership changes in GMS. A client only need to query the GMS to get the reference to an available replica, thus makes the direct connection with that replica. For a primary-backup group, the GMS will always return the primary member so that all subsequent requests can be directed to the primary. For a primary-peer group, GMS returns the reference to the least loaded replica. Thus, each replica is connected with roughly even number of clients.

4. The Tool Components

In essence, the tool for constructing a replicated service system based on the PPRS includes three components: the replica group interface definition language (RGIDL), the RGIDL preprocessor and the PPRS runtime system.
The Rigid Language. The RIGID language is used to specify a replica group and a service interface embracing a set of operations exported to clients. Table 1 shows the RIGID syntax in EBNF.

<table>
<thead>
<tr>
<th>Term</th>
<th>EBNF</th>
</tr>
</thead>
<tbody>
<tr>
<td>ReplicaDef</td>
<td><code>replica' Id </code>{' GroupDef <code>{' InterfaceDef </code>{'}<code>}</code></td>
</tr>
<tr>
<td>GroupDef</td>
<td><code>group' Id </code>{' PB <code>{'PP </code>{' GroupConfig <code>{'}</code>}<code>}</code></td>
</tr>
<tr>
<td>GroupConfig</td>
<td>MemberList GMSRef Propagation</td>
</tr>
<tr>
<td>MemberList</td>
<td><code>sites' </code>{' [IPNameList] <code>{' PortNum </code>{'}<code>}</code></td>
</tr>
<tr>
<td>IPNameList</td>
<td><code>[IPName </code>{'}<code>]</code></td>
</tr>
<tr>
<td>GMSRef</td>
<td><code>GMS' </code>{' IPName <code>{' PortNum </code>{'}<code>}</code></td>
</tr>
<tr>
<td>Propagation</td>
<td><code>propagation' </code>{'every' Number <code>[ms]</code> <code>{'}</code></td>
</tr>
<tr>
<td>PortNum</td>
<td><code>Number</code></td>
</tr>
<tr>
<td>InterfaceDef</td>
<td><code>interface' Id </code>{' [OperationDef] <code>{'}</code></td>
</tr>
<tr>
<td>OperationDef</td>
<td><code>OpType</code> <code>{' OrderConstraint} JavaMethodDefinition </code>{'`</td>
</tr>
<tr>
<td>OpType</td>
<td><code>query' </code>{'update'</td>
</tr>
<tr>
<td>OrderConstraint</td>
<td>`causal'</td>
</tr>
<tr>
<td>IdList</td>
<td><code>[Ident </code>{'}<code>]</code></td>
</tr>
<tr>
<td>Ident</td>
<td><code>[a-zA-Z]+[0-9]*</code></td>
</tr>
<tr>
<td>Number</td>
<td><code>[0-9]+</code></td>
</tr>
<tr>
<td>IPName</td>
<td><code>[Ident </code>{'}<code>]</code></td>
</tr>
</tbody>
</table>

Table 1. The RIGID Syntax in EBNF

The replica group declaration defines a group name which has to be a unique identifier under the GMS. A group can be specified as either PP or PB to distinguish a primary-peer group from a primary-backup group. A group can have an initial site list on which the replicas will be running. Propagation frequency can be specified in two ways: (1) Every number of requests; (2) A time interval.

The service interface defines a set of operations that the server exports to clients. Each operation can have two specifications: (1) operation type: query or update; (2) ordering constraint: causal, total or tocausal (total=causal, if the operation is defined as an update operation). The rest of the operation is defined by JavaMethodDefinition which follows the syntax of Java method definitions.

The Preprocessor. With a given RIGID specification file, the preprocessor generates three sets of files: (1) the service interface file containing a set of Java methods; (2) client driver (client object) and client proxy files to form the client side program; and (3) server driver (service implementation object) and server surrogate files to form the replica side program.

The PPRS Runtime Library System. The runtime system supports the execution of both client programs and replica programs. The design principle for the runtime system is to layer objects so that high layer objects are supported by lower layer objects. The runtime system is developed on the basis of Java networking package and itself is a package (named PPRS package) consisting of three layers of objects: utility objects (at bottom layer), mini-protocol objects (at middle layer), and root proxy and root surrogate (at top layer).

Utility objects manage basic resources, such as the replica group membership, queues for buffering incoming and outgoing messages, communication channels between clients and replicas, etc. Mini-protocol objects implement three protocols for managing membership changing events: a replica leaving voluntarily, a replica crashing, and a new replica joining. Root proxy object is supposed to be inherited by all service-and-group-specific client proxies. Root surrogate object is to be inherited by all group-specific surrogates.

5. An Example of a Primary-Peer Replication

In this section, we will give an example based on the primary-peer group. The replicated service is a room booking service (RBS). The RIGID file `RBS.idl' defines a 3-replica group and the service interface as:

```java
replica RBS { group RBSgroup:PP { ...
```
Figure 2. The files generated by RGIDL preprocessor (rgidigen)

The specification declares a primary-peer group named RGSgroup. The primary will be running on the machine "bofus", and two-peer replicas will be running on "haleth" and "durin". Replicas run at the same port number of 1200. The GMS server is running on "alice" at port 1000. The propagation frequency is every 1 request. The files generated by the preprocessor are depicted by Fig. 2. Both the primary and peers use the same code fragments, but start execution in different roles: java RGSser 0 — the primary, and java RGSser 1 — the peer.

6. Concluding Remarks

In this paper we have presented a tool architecture aimed at automating the development of distributed replication systems. The design ideas mimic the principles of the RPC system. The implementation is based on Java language, fully object-oriented designs are enforced throughout the development of the tool. The runtime library is constructed as a Java package with layered objects [6].

By using this tool, the developer of replicated service only needs to concentrate on the implementations of service programs and client interface programs, and leaves the replication control protocol to be generated by the tool. This shortens the development life-cycle and reduces the design complexity significantly.

References
Automating the Construction of Service Replication systems

Li Wang  
School of Comp. and Math.  
Deakin University  
Geelong, Vic 3217  
Australia  
email: liwang@deakin.edu.au

Wanlei Zhou  
School of Comp. and Math.  
Deakin University  
Geelong, Vic 3217  
Australia  
email: wanlei@deakin.edu.au

Abstract  
Service replication is a key to providing high availability, fault tolerance and good performance in distributed systems. However, building a service replication system is a difficult and complex task. Traditionally, the remote procedure call (RPC) system is a common tool for building client/server systems. However, the RPC does not provide sufficient support in building service replication systems. This paper proposes a toolkit architecture aimed at automating the construction of service replication systems. The toolkit includes a language for describing a replication system, a preprocessor that generates relevant modules to form client programs and replicated server programs, and a runtime system that supports the executions of the replicated servers. The replication transparency is achieved through a separate group management service.

Keywords: Service Replication, RPC System, Primary-Backup Scheme, Active Replication.

1 Introduction  
Replication in general is an important approach to increasing the availability, achieving fault-tolerance, and improving the efficiency of a system. Distributed replication is a common means by which a system can provide continuous services in the presence of component failures. The service availability is enhanced by replicating data/software on multiple sites that fail independently. The failure of one server can be tolerated by its replicated servers (replicas). Better performance is also achieved by replicas executing client requests concurrently.

Replication schemes generally fall into two styles, the primary-backup scheme [2, 4] and the active replication scheme [5, 4]. In the primary-backup approach, it designates one replica as the primary and others as backups. Only the primary accepts client requests and generates replies, the backups stand by and wait for state changes propagated by the primary from time to time.

The active approach is based on the architecture that every replica receives requests. A query request is sent to one replica, whereas an update request is sent to and will be executed by all members of the group eventually. Active replication is an advanced scheme and thus relatively more complex scheme than the primary-backup scheme.

Traditionally, the remote procedure call (RPC) system helps building the client/server systems very well when the server is a single entity. However, when it comes to build a replicated server system, the RPC system does not provide sufficient support. The reason is that the RPC system only provides point-to-point communications between the client and the server. Whereas in the replicated server system, the client has to communicate with a group of replicas, and replicas have to communicate with other replicas intensively upon each update request sent out from a client to guarantee the data consistency among replicas.

There are also a number of toolkits developed by various research groups, such as Isis [1], GARF [3], and the like, claimed to support building replication systems. However, these toolkits only provide library-level functions and protocols, programming based on these toolkit is still manual and programmers have to have intimate knowledge about the functions and protocols provided by the toolkits.

The purpose of this paper is then to develop a toolkit architecture aimed at automating the developing procedure of replication systems. Based on a general integrative replication scheme that accommodates both primary-backup and active replication approaches, the toolkit includes an interface definition language for defining a replica group, a preprocessor that generates a set of modules for constructing client and replica programs, and a runtime system that supports the executions of client and replica pro-
grams. This work is the continuation of our previous work [6, 7]. The implementation is based on Java and fully object-oriented designs are enforced throughout the toolkit development.

The rest of the paper is organized as follows. Section 2 presents the execution model. Section 3 describes the tool architecture. Section 4 gives an example based on the primary-backup replication. Section 5 concludes the paper.

2 The Object-Based Execution Model

We propose a general replication scheme that can accommodate two styles of replication schemes. Fig. 1 depicts this scheme. The scheme is named as primary-peer replication scheme (PPRS). The scheme is based on active replication, but one replica is designated as the primary, others as peers. A primary-backup group is configured simply by directing all requests to the primary of the group and letting peer members (now backups) only receive propagations.

![Diagram of the Primary-Peer Replication Scheme]

Figure 1: The Primary-Peer Replication Scheme

From Fig. 1, we can extract two basic system components, namely client and replica. To separate concerns, we use an execution model that mimics the RPC system where a client is supported by a client stub and a server is supported by a server stub. In our executional system, a client is supported by a client proxy and a replica (replicated server) is supported by a replica surrogate.

In essence, an executable replication system consists of three types of software entities: client, replica, and the group management server (GMS). Fig. 2 depicts the relationship among these entities. These software entities are represented and implemented by objects.

2.1 Client

A client is composed of two objects: the client object and the client proxy. The client object is an application level program. It often provides a user interface (such as a graphic user interface) to access operations provided by a distributed server.

From a client object point of view, there should be no difference between a replicated server and a non-replicated server (single server), nor is there any difference to use it. This illusion is provided by the client proxy. The client proxy implements the functionality of dealing with a replica group and makes a replicated server group appear to be a single server.

2.2 Replica

The functionality of a replica is two-fold: providing a set of operations, and coordinating with other replicas to implement the replication control protocol. The replication protocol of the PPRS is based on the update ordering model where each update operation has an ordering constraint. An ordering constraint can be specified in terms of causal, total, and total-cause [1]. Therefore, a replica is split into two objects, a service implementation object (referred to service object) and a replication management object (referred to surrogate object). The service object implements the set of operations that are exported to clients. The surrogate object implements the replication control protocol on behalf of the corresponding replica. Surrogates coordinate with each other to provide a unified system behavior. By separating the service implementation from the replication control protocol, the system is enhanced with modularity and flexibility.

2.3 Group Management Service

In practice, when a crashed replica leaves the system, a new replica may be added to the system from a different site. Thus, the membership of a replica group is dynamic. We use a group management service (GMS depicted in Fig. 2) as a bridge between clients and the replica group so that any membership change can be passed on to the clients. The GMS is set up separately on a stable site. A replica group is registered with the GMS under a unique group name, any change of the membership is sent to the GMS by the primary. Clients only need to query the GMS to get the reference to an operational member, then make the direct connection with that member. The GMS manages multiple replica groups for the reason that, in real life, there maybe multiple critical objects being replicated in a distributed system. The interface provided by the GMS should include at least the following functions:

- `createGroup()`. To create a group.
- `addMember()`. To add a member into a group.
- `deleteMember()`. To remove a member from a group.
bind(). To bind a client with a member.

When a group of replicas is created, the primary of the group registers the group to the GMS (invoking createGroup()). The primary is also updating the GMS with any membership change (invoking addMember() or deleteMember()). bind() is invoked by clients to get a reference to a currently available member of the group. For a primary-backup group, the bind() will always return the primary member so that all subsequent requests can be directed to the primary. For a primary-peer group, all members form a ring. Upon receiving a binding request, the GMS finds the member with the least number of connected clients and returns that member to the client. By doing so, each member is connected with roughly even number of clients, thus, load is split over replicas evenly.

3 The Tool Architecture

In essence, the tool for building replication systems has three parts: the replica group interface definition language (RGIDL), the RGIDL preprocessor, and the RGIDL runtime system.

3.1 The RGIDL Language

The RGIDL language is used to specify a replica group and the service interface to clients. Table 1 shows the RGIDL syntax in EBNF. A replication system is defined by two parts, a group definition part and a service interface definition part.

**Group part.** A replica group is specified by the group name, group style (primary-backup or primary-peer), initial members (if there is any), propagation frequency, and the GMS reference.

**Service Interface part.** It defines a set of operations exported to clients. Each operation has the same syntax as that of the Java method, beside, two specifiers, OpType and OrderConstraint can be added in front of each operation. OpType specifies a query or an update operation, and OrderConstraint specifies the ordering constraint.

3.2 The Preprocessor

According to a given RGIDL specification file, the preprocessor generates three sets of files: (1) the service interface file containing the set of operations exported to clients; (2) client driver (client object) and client proxy files to form the client side program; and (3) server driver (service implementation object) and server surrogate files to form the replica side program.

3.3 The RGIDL Runtime System

The runtime system gives support in executing both client programs and replica programs. The design principle for the runtime system is to layer objects so that higher layer objects are based on and supported by the lower layer objects. Fig. 3 depicts the layers of the runtime object model.

The runtime system is developed on the basis of Java networking package and itself is a package (named PPRS package) consisting of a collection of objects: utility objects, mini-protocol objects, root proxy and root surrogate.

![Figure 3: The Object Layers](image)

- **Utility objects:** Utility objects manage basic system resources, such as the local group view, messages transmitted among system entities (clients...
| ReplicaDef                      | := | 'replica' Ident '{' GroupDef (InterfaceDef)+ '}' |
| GroupDef                       | := | 'group' Ident '{' 'PB'|'PP' '{' GroupConfig '}' |
| GroupConfig                    | := | MemberList GMSRef Propagation |
| MemberList                     | := | 'sites' '{' [IPNameList] '}' 'PortNum' '}' |
| IPNameList                     | := | (IPName '+')* IPName |
| GMSRef                         | := | 'GMS' ':' IPName ':' PortNum '}' |
| Propagation                    | := | 'propagation:' 'every' Number ['ms'] '+' |
| PortNum                        | := | Number |
| InterfaceDef                   | := | 'interface' Ident '{' [OperationDef]+ '}' |
| OperationDef                   | := | [OpType] [OrderConstraint] JavaMethodDefinition ',' |
| OpType                         | := | 'query' '|' 'update' |
| OrderConstraint                | := | 'total' '|' 'causal' '|' 'tcausal' |
| IdentList                      | := | [Ident,]* Ident |
| Ident                          | := | {a-zA-Z0-9}* |
| Number                         | := | [0-9]+ |
| IPName                         | := | [Ident,]* Ident |

Table 1: The RGIDL Syntax in EBNF

and replicas), the system information of sites and port numbers, communication channels among replicas, queues for incoming and outgoing messages, and so on.

- **Mini-protocols:** In the event of a replica crashing, leaving voluntarily, or a new replica joining the group, certain protocols are to be executed to handle the situation. These protocols are implemented as mini-protocol objects. Currently we have three mini-protocols corresponding to three types of membership changing events: StateTransfer, CrashAtomicity and VoluntaryLeave.

- **Root proxy:** The proxy is an extended client side stub that handles communications with the GMS and the replica group. The proxy of the runtime library is the root proxy that is to be inherited by all specific client proxies.

- **Root surrogate:** The surrogate object is an extended server side stub which handles the communications with proxies and peer surrogates. The replication control protocol is embedded in the surrogate. The surrogate of the primary replica is responsible for updating the GMS with any membership change. The surrogate of the runtime library is the root surrogate to be inherited by all specific replica surrogates. The surrogate is designed by employing the following ideas:

  1. Communications are based on reliable TCP/IP channels. Reliable TCP/IP streams are created between members of a replica group, and also between members and clients. (2) Multi-threading is used to improve the performance of the surrogate. Multi-threading has made the constructing of the surrogate easy. It solves the problem that a client can make connection to the replica group at any time, and clients that are connected to the same member can send requests in parallel. (3) The surrogate object is a composite and an aggregate object of lower-level objects.

4 An example

In this section, we will give an example of replicating a host name service (HNS) in the primary-backup style. Without loss of generality, we assume one backup is employed.

Suppose we need a simple host name service that stores mappings between symbolic host names and their IP addresses of a local domain. To be able to provide continuing HNS in the presence of a crash failure, we replicate the HNS to a primary-backup group. Assume the following operations are provided by the HNS server:

1. `addHost (String HostName, String IPAddress)` An update operation which adds a new host to the local domain.
2. `updateHostName (String OldHost, String NewHost)` An update operation that changes an old host name to a new one.
3. `deleteHost (String HostName)` An update operation that removes a host from the local domain.
4. hostName (String InetAddress). A query that returns the host name that has the IP address.
5. InetAddress (String HostName). A query that returns the IP address of the host.

The RGDIDL file HNS.idl defines the group and the service interface:

```java
replica HNS {
  group HNSgroup:PB {
    sites: {befur.cm.deakin.edu.au,
            gollum.cm.deakin.edu.au:1040
    GMS: alice.cm.deakin.edu.au:1050
    propagation: 1
  }
  interface HNS {
    update public void addHost
           (String HostName, String InetAddress);
    update public void updateHostName
           (String OldHost, String NewHost);
    update public void deleteHost (String HostName);
    query public String InetAddress (String HostName);
    query public String HostName (String InetAddress);
  }
}
```

The specification declares a primary-backup group named as HNSgroup. The primary will be running on machine befur.cm.deakin.edu.au, and the backup will be running on gollum.cm.deakin.edu.au, both are running at the same port number of 1040. The GMS server is running on alice.cm.deakin.edu.au at port 1050. The propagation frequency is every 1 request from the primary to the backup. Fig. 4 shows the procedure of using the preprocessor. The preprocessor generates a set of files according to the RGDIDL definition file, HNS.idl:

- **HNS.java** — the service interface file. It defines the set of operations.

  ```java
  public interface HNS {
    public void addHost
           (String HostName, String InetAddress);
    public void updateHostName
           (String OldHost, String NewHost);
    public void deleteHost(String HostName);
    public String InetAddress(String HostName);
    public String HostName(String InetAddress);
  }
```

- **HNSProxy.java** — the client proxy file. The file contains the HNS proxy object which is an extension to the root proxy in the runtime system and implements the HNS service interface:

  ```java
  public final class HNSProxy extends PPRS.Proxy implements HNS {...}
```

The **HNSProxy** first queries the GMS to get the reference to a primary, then it makes the direct connection to the primary. When the primary is down, HNSProxy queries the GMS to get the backup's reference and switches to the backup.

- **HNScli.java** — the client driver file. The file contains the client object that instantiates HNSProxy and then invokes any operation provided by the HNS server.
- **HNSser.java** — the server surrogate file. The file contains the HNS surrogate which is an extension to the root surrogate of the runtime system:

  ```java
  public class HNSSurrogate extends PPRS.PBSSurrogate {...}
```
- **HNSSer.java** — the server driver file. The file contains the implementation of the HNS interface and the server driver that instantiates and starts up the HNS surrogate.

Two executable files HNScli.class and HNSser.class are generated after the compilation. Both the primary and the backup use the same code fragments of service object and server surrogate but start the execution in different roles: java HNSser 0 denoting the primary and java HNSser 1 denoting the backup. When the primary and the backup have established the connection to each other and are ready to accept requests, clients can be started to send requests.

**usage:** HNSser 0 1 //start the primary/backup
**usage:** HNScli //start the client

### 4.1 Performance

The performance study is conducted within a local subnet (10M Ethernet) of Sun Sparc stations and using Java JDK 1.2 version. The tests use one client making a synchronous (blocking) request. The response time is measured by the time delay between sending a request and receiving the reply. The response time shown in table 2 is the mean time of 100 same requests.

<table>
<thead>
<tr>
<th>group type</th>
<th>response time (ms)</th>
</tr>
</thead>
<tbody>
<tr>
<td>primary</td>
<td>4</td>
</tr>
<tr>
<td>primary-backup</td>
<td>11</td>
</tr>
<tr>
<td>switch-over</td>
<td>18</td>
</tr>
</tbody>
</table>

**Table 2: The Primary-Backup Performance**

Three simple experiments are conducted: (1) **Non-replicated server** — the primary only. The response
time (4ms) is measured between sending a query request and receiving the reply from the primary. (2) The primary-backup group. The test is conducted by sending an update request. The primary propagates the request to the backup before replying to the client. The response time (11ms) shows that introducing a backup results in a longer response time. (3) Primary crashing. During an update request, the primary crashes, the client has to switch to the backup. The switch-over procedure includes: contacting the GMS, making the connection with the backup, and re-sending the request. The response time is measured between the first request sent to the primary until receiving the reply from the backup. The test shows the worst response time (18ms).

The above simple performance study only reflects the response time of a primary-backup group involving one backup. Due to the paper size limit, extensive performance studies on a primary-backup group containing multiple backups and a primary-peer group with different update ordering constraints are not included here. We are addressing them in a separate paper.

5 Concluding Remarks

This paper has presented a toolkit for automating the development of distributed replication systems. The design mimics the RPC systems, a proxy is used to support the client program, a surrogate is used to support the replica program. The RGDIL language is proposed for defining a replicated server group and a set of remote operations. A preprocessor generates the relative proxy and surrogate objects to form the client and the replica side programs.

The implementation is based on Java language. The runtime system is built on the basis of Java virtual machine. The simple example has shown how to use the toolkit.

We also introduced a group management service for managing dynamic replication groups so that a new replica can be introduced to the system at any time and on any site. This greatly improves the flexibility of the replication system.

References


The design and implementation of an active replication scheme for distributing services in a cluster of workstations

Li Wang a, Wanlei Zhou b,*, Weijia Jia c

a Phoneware Communication Systems, Bentwood East, Vic. 3151, Australia
b School of Computing and Mathematics, Deakin University, Clayton, Vic. 3168, Australia
c Department of Computer Science, City University of Hong Kong, Hong Kong

Received 23 March 2000; received in revised form 1 July 2000; accepted 21 September 2000

Abstract

Replication is the key to providing high availability, fault tolerance, and enhanced performance in a cluster of workstations (COWs). However, building such a system remains as a difficult and challenging task, mainly due to the difficulty of maintaining data consistency among replicas and the lack of easy and efficient tools supporting the development procedure. In this paper we propose an active replication scheme in which data consistency can be maintained. Based on the active replication scheme, we present an object-oriented design pattern and a constructing tool to simplify the design and implementation of service replicas in COWs. © 2001 Elsevier Science Inc. All rights reserved.

Keywords: Distributed replication system; Distributed object; Cluster of workstations; Fault-tolerance; Group communication

1. Introduction

Providing reliable and efficient services are the primary goals in designing a distributed system running on a cluster of workstations (COWs). Nowadays, we have seen a trend in distributed system design to shift from two-tier architectures or even monolithic architectures to three-tier or even n-tier architectures supported by the client/server model (Goscinski and Zhou, 1999). This means that a lot of services of a distributed system originally provided by a single piece of software are moved out of the kernel, forming individual servers. It is then crucial to guarantee that these servers will provide reliable and efficient services.

Service replication is a common means by which critical software and/or data are duplicated on a cluster of workstations. The goal of designing a replication system in general is to provide resilience and availability in the presence of failures while preserving one-copy consistency among all replicas. Service replication also provides a way of achieving better performance by distributing client requests to replicated servers to avoid the bottleneck in a single server environment.

Replication schemes generally follow two streams, namely primary-backup and active replication (Powell and Verissimo, 1991; Guerraoui and Schiper, 1997). Active replication is also referred as state machine approach (Schneider, 1990). There are various hybrids of above two replication models to suit application-specific requirements, such as the coordinator-cohort (Birman and Glade, 1995) and the semi-active replication (Powell and Verissimo, 1991). This paper deals with the active replication schemes. We propose an active replication scheme where data consistency is defined by all operational replicas delivering the same set of update requests.

An active replication system uses the group communication concept. A lot of research has been done on group communication services. Isis (Birman, 1993) provides a complete toolkit based on process groups and the virtual synchrony concept for reliable distributed computing. Guerraoui and Schiper (1997) described how group services can support primary-backup and active replications. GARF (Guerraoui et al., 1995) is a distributed object-oriented environment aimed at supporting the design and programming of reliable distributed applications through an extensive library of generic objects. The shortcoming of these systems is the lack of flexibility and the complexity of learning and use. The common object request broker architecture
(CORBA) (Orfali and Harkey, 1998) is a platform generally for developing client-server distributed object computing model. However, fault tolerance was not a major concern in CORBA design.

This paper is aimed to promote distributed object methodologies to bring flexibility and modularity into distributed replication systems. Based on the active replication scheme, we propose an object-oriented design pattern and a constructing tool to simplify the design and implementation of distributed replications on COWs. Java is used in this paper as the target language for system implementation. Through the experience of authors, it is indeed proven that Java is a neat and easy-to-work-with high-level language for developing programs running on COWs. Java-like pseudo code is used throughout this paper to demonstrate various design ideas.

The rest of the paper is organized as follows. Section 2 describes the active replication scheme. Section 3 presents the group services to facilitate the design of the active replication scheme. Section 4 gives an implementation overview for the object-oriented design pattern. Section 5 describes the architecture of the constructing tool. Section 6 presents some results in performance evaluation. Section 7 summarizes the paper.

2. The active replication scheme

We model a distributed replication system as a group of replicas (or members, replica and member are used interchangeably in this paper) located on a cluster of workstations, only one replica at a workstation. The functionality of a replica is twofold, providing a set of services and implementing the replication policy. The replication system as a whole should appear as a single service provider to clients. Fig. 1 depicts our active replication scheme. The model has the following components:

- Client: A client (c_i) is connected to only one member (R_j) of the group at a time.
- Replica (member): A replica or member is responsible for processing and replying the client's requests. An update request received at a member is multicast to all other members. A query request is delivered and processed locally.
- The replica group and its data consistency policy: Replicas are organised as groups and the data consistency among group members is defined by all operational members delivering the same set of updates in total order. This policy is guaranteed by the group communication services described in Section 3.
- The name server (NS): The NS (described later in this section) is a stand-alone server that runs separately from either clients or replica groups. It provides services for bridging the clients and replicas.

A replica is designed as an integration of two objects, the service implementation object (service object in short) and the replication management object, to separate the services provided and the replication policy adopted. The service object implements a set of services that are exported to users. The service object is purely interested in providing correct services without being involved in any matters like where it should be located or whether it will be replicated. It is a distributed object that can be freely moved onto any workstation. The granularity of service object could be from as large as a database manager, to as small as a queue manager. Services provided are categorized as either queries or updates. A query does not change the state of the service object but an update does.

The replication management object plays a role of being a surrogate to a service object. A set of replication management objects forms the object group that implements replication policies. Here we give Java-like skeleton codes for the service object (ServiceImpl), the replication management object (Surrogate), and the interface between them (ServiceInterface).

```java
public interface ServiceInterface { // implemented by any service object.
    public String upCall(String request);
}

class ServiceImpl implements ServiceInterface {
    // service provider.
    private variables;
    public service_1 () {} 
    ...
    public service_n () {}
    public String upCall (String service-name with-arguments) {
        switch (service-name) {
            case service_1:
                service_1(tho-arguments);
            ...
            case service_n:
```
service_a(the-arguments);
}
}

class Deliver {...} //deliver client requests.
class Surrogate {//implementing the replication policy.
    private Deliver deliver = new Deliver();
    private ServiceInterface service;
    ...
    public Surrogate (ServiceInterface service) {} 
    public void handleRequests() {
        while (true) {
            service.upCall(deliver.deliverReq());
        }
    }
    public static void main() {
        ServiceImpl service = new ServiceImpl();
        Surrogate replica = new Surrogate (service);
        replica.handleRequests();
    }
}

Through the interface upCall(), any service object implementing the upCall() can be integrated with the underlying replication system (surrogate object) without any internal modifications involved in both objects. By adopting this interface, different service objects are able to plug-and-play. If a service object needs to be upgraded, the surrogate is not affected, and vice versa.

The replica group is designed to be dynamic in the sense they do not operate on a static set of workstations, rather, any workstation can be used. The group membership of the replication system is changed under three kinds of events, join, leave, and fail. A join event is issued by a new member joining the group, or by a rejoining member. A leave event happens when a member leaves voluntarily. A fail event is due to a member's crash failure.

An NS is established to sit between clients and the group. We assume the NS is installed at a stable workstation. By doing so, the replication group is isolated from clients; clients do not need to keep the membership of the group, rather only have to know the address of the NS and the symbolic name of the replication group.

Upon a request to the NS for a connection to a replicated server, a client is bound with a member of the group by the NS. Then the client makes the direct connection to that member which becomes the accessing point of the replicated service. When that member becomes unavailable, the client asks the NS for the next available member. Membership changes are informed by the replication system to the NS. The dynamic membership of the group is passed through the NS to clients in a dynamic and transparent way.

The NS can employ some load balancing policy to assign group members to clients so that loads of group members are roughly even. We form a ring of operational members in the same server, a client is allocated with the next member in the ring. Thus the clients are evenly allocated to members of the group.

Here we give the NS interface. This interface can be used to manage multiple replication groups.

```java
public interface NameService {
    void createGroup(String[] members, String groupname)
        throws GroupNameIsAlreadyUsedException;
    void addMember(String groupname, String member)
        throws GroupNameNotFoundExpection,
            MemberNotFoundExpection;
    void deleteMember(String groupname, String member)
        throws GroupNameNotFoundExpection,
            MemberNotFoundExpection;
    Service bind(String groupname) throws
        GroupNameNotFoundExpection,
        GroupIsNotEmpty;
}
```

3. Group communication services and replication consistency

Without considering failures, the implementation of group services is trivial. However, group services need to be fault-tolerant to provide expected services in the face of failures.

Any group member may fail then becomes unavailable. A failure can be caused by a number of reasons. (1) Voluntary-exit. A member exits voluntarily due to, such as lacking of resources, being switched off by the system administrator, etc. In this case it can restart at a well-defined state. (2) Process crash. A member process stops the execution due to various reasons, such as being killed deliberately. It may not be able to restart at a well-defined state. (3) Site crash. The workstation that a member is running on has crashed. The member definitely cannot restart at a well-defined state. We do not consider network partitioning failures (Millar-Smith and Moser, 1998) here. This problem is dealt with in another paper (Zhou and Jia, 1999).

The basic idea of the multicasting atomicity protocol is to let operational members exchange (re-multicast) their last received message from the failed member. Let \( G = \{ g_1, \ldots, g_n \} \) be the members of the group. At \( g_i \), it keeps a vector of the latest received multicast from every
other members, \( \text{last.mcast}[g_k] = m^k_0 \), where \( k = 1, \ldots, n \). In the absence of failures, upon reception of \( m^0_0 \), \( g_i \) updates its vector, i.e. \( \text{last.mcast}[g_i] = m^0_0 \).

When a member, say \( g_x \), first detects the failure of \( g_x \), it initiates the multicasting atomicity protocol by multicasting its \( \text{last.mcast}[g_x] \), that is, \( g_x \) multicasts a \( m_{\text{atom}}(g_x, m^0_0) \) indicating \( g_x \) has failed and attaching the latest received \( m^0_0 \) from \( g_x \). Upon receiving \( m_{\text{atom}}(g_x, m^0_0) \), a member updates its \( \text{last.mcast} \) vector according to the following algorithm:

**Multicasting atomicity protocol**

*At each operational member \( g_x \):

On receiving \( m^0_0 \):

\[
\text{last.mcast}[g_x] := m^0_0.
\]

On detecting the failure of \( g_x \):

Multicast \( m_{\text{atom}}(g_x, \text{last.mcast}[g_x]) \)

On receiving \( m_{\text{atom}}(g_x, m^0_0) \):

\[
\text{if} \ \text{last.mcast}[g_x] = m^0_1 \\
\text{then update last.mcast}[g_x] := m^0_1; \\
\text{else if the case of last.mcast}[g_x] = m^0_0 \text{ or last.mcast}[g_x] = m^0_1 \\
\text{ignore} m_{\text{atom}}(g_x, m^0_0); \\
\end{if}
\]

\[
\text{if} \ g_x \text{ has not sent a} m_{\text{atom}} \\
\text{then Multicast} m_{\text{atom}}(g_x, \text{last.mcast}[g_x])
\]

During the execution of the multicasting atomicity protocol, it is likely there might be other failures. For brevity, let \( g_1 \) be the first one who fails, then \( G \) becomes \( \{g_1, \ldots, g_x\} \), and \( m^0_1 \) is the message unstable in \( \{g_1, \ldots, g_x\} \). Multicasting atomicity protocol is performed by members in \( \{g_1, \ldots, g_x\} \) to propagate \( m^0_1 \). During the execution, say, \( g_1 \) holds the \( m^0_1 \), it then multicasts \( m_{\text{atom}}(g_1, m^0_1) \). If \( g_1 \) fails during the multicasting, it leaves \( G = \{g_2, \ldots, g_x\} \). Thus \( m^0_1 \) has to be recovered in \( \{g_2, \ldots, g_x\} \). This procedure may go on if there are further failures until \( G = \{g_x\} \). No matter whether \( g_x \) has got \( m^0_0 \) or not, the atomicity is satisfied.

The multicasting atomicity protocol will not cause a multicast storm. First of all, if there is only one failed member, say \( g_x \), and \( g_i \) is the member who first detected \( g_x \)'s failure, then \( g_i \) will use the same message to multicast the message \( m_{\text{atom}}(g_x, \text{last.message}[g_x]) \) to all members. If \( g_x \)'s \( \text{last.message}[g_x] \) is the latest message from \( g_x \), then the multicasting stops. Otherwise, only the members containing the latest \( \text{last.message}[g_x] \) will perform the multicast once. If there are multiple member failures during the multicasting atomicity protocol, the protocol will stop when the unstable messages of all the failed members have been multicasted, as described in the previous paragraph.

Membership is the information about a group that shows who is in the group. The membership is changed by a member's jointleave/fail event. In essence, the membership is replicated at each member and also the NS, so a multicast can be sent to those who are currently operational, and also clients can connect to new members through the NS.

The membership is represented by a view consisting of a set of \( \{g_1, \ldots, g_x\} \) and is encapsulated by operations \( \text{leaveGroup()} \) and \( \text{joinGroup()} \). Members start with the same local view, \( \text{view}_0 \). Let \( \text{view}_i, i = 0, 1, \ldots \) be the successive views of the group, \( \text{view}_0 \) and \( \text{view}_{i+1} \) are different only by an addition or a deletion of one member. A view update message \( \text{mupdate/mupdateAtom} \) is multicasted to members in \( \text{view}_i \cap \text{view}_{i+1} \), changing \( \text{view}_i \) to \( \text{view}_{i+1} \).

View changes are delivered in total-order at all members. There are generally two types of algorithms used to implement total ordering. One is to generate a unique sequence number (USN) for each request. Thus, requests can be executed in a unique sequential order group wide. The other one is the token-ring algorithm (Jia, 1997). The USN algorithm has two approaches as well: centralised approach (Kahan and Tangenbaum, 1989) and distributed approach (Scheidler, 1990). We adopt the simpler approach—the centralised sequencer for implementing the total ordering of requests. THAT IS, the total ordering is accomplished by designating a member as the coordinator. A jointleave/leave request is sent to the coordinator from where it is multicasted to all other members. If the coordinator fails, the new coordinator is selected by an algorithm, for instance, the oldest member in the group.

**Member joining.** A joining event is a concurrent event with multicasting events among group members, a scenario depicted by Fig. 2. Where \( g_x \) is the new member joining the group. At the same time, a multicast \( m_i \) is sent to current view \( \{g_1, g_2, g_3\} \), which is delivered at \( g_2 \) and \( g_3 \) before \( m_{\text{join}} \), but after \( m_{\text{join}} \) at \( g_1 \). Suppose \( g_1 \) is the coordinator, it transfers its state to the new member at the instant of receiving \( m_{\text{join}} \). Later on, when \( g_1 \) delivers \( m_i, g_4 \) will miss it.

![Fig. 2. Member joining scenario.](image-url)
Virtual synchrony on membership service. The virtual synchrony model developed by Isis project defines the membership atomicity, which is a property that messages sent in view have to be delivered in the same view. That is, a \( m_{\text{new}} / m_{\text{old}} \) is delivered at all members of view when messages sent in view are group-wide stable. Membership atomicity is implemented by the flush protocol (Birman, 1993).

Member leaving. Conceptually, the flush protocol is not needed when a leaving event \( (m_{\text{new}} / m_{\text{old}}) \) happens, since state transfer is not involved. But flush protocol can serve as a checkpointing mechanism that a member leaving voluntarily can have a consistent state with the group at the instant of departure. Later, this member can rejoin the group by catching up missed operations. The catch-up might be a more appropriate solution than state-transfer in the situation where the group manages a large volume of data.

When dealing with a member leaving by crash, both multicasting atomicity protocol and flush protocol have to run. Multicasting atomicity protocol stabilizes the last multicast of the failed member which should be delivered to all members before the membership is updated. Therefore, these two protocols can be combined in one go by letting \( m_{\text{new}} \) carry the unstable multicast of the failed member.

The membership consistency between the group and the NS. A member designated as the coordinator is responsible for updating the NS right after its local view is updated. Hence, the NS executes the same set of membership updates in the same order as the group.

4. Implementation issues

There are three components in the system: replicas, clients, and the name server. Fig. 3 depicts the relationship of these components. A replica is composed of two objects, the service object and the surrogate object. We have presented the skeleton codes for the service object and the name service interface in Section 2. The most complex object is the surrogate object. This section is focused on the internal design of the surrogate object, which is a coherent integration of many other objects.

A member is fully connected with all other members. The replicated server group can be created in a number of ways: (1) The group is created by taking a list of members as the input. (2) The group is created with empty member, then each member joins the group.

The surrogate has the following state variables:

\[
\text{member.state} = \{ G, \text{InQ}, \text{OutQ}, \text{last.mcast}[G] \}.
\]

\( G \) represents the local view of the group membership. \( \text{InQ} \) stores all incoming messages from both clients and other members of the group. Requests are delivered in FIFO ordering. \( \text{OutQ} \) stores outgoing replies. \( \text{last.mcast}[G] \) is used by the multicasting atomicity protocol.

Each surrogate runs multiple threads to handle different matters. There are three types of threads, client-connection thread, multicast-reading thread, and the main thread. A member starts a thread for accepting clients’ connections. This thread spawns a child thread for each client connection. A child thread is used to read the client requests, deposit them to \( \text{InQ} \), get replies from \( \text{OutQ} \), and write them back to the client. A multicast-reading thread is setup to read all propagated messages from peer members of the group. The main thread controls the whole process of replication management by delivering requests from \( \text{InQ} \), issuing up-calls to the service object, and putting them to \( \text{OutQ} \).

Messages are formed by using a three-tuple format:

\[
\text{Message} :: = (\text{type}, \text{mid}, \text{message-content});
\]
\[
\text{Type} :: = \text{'u'/'q'/'p'/'j'/'f'/'h'/'r'};
\]
\[
\text{mid} :: = \text{sender-network-address:message-sequence-no};
\]
\[
\text{message-content} :: = \text{the-content-of-the-request};
\]

Messages are typed in order to direct different actions.

A surrogate is an integration of many auxiliary objects. Here we describe in detail how the surrogate object is constructed.

- class Group. A group object manages the local view of the operational members. Basically, it provides two methods, \( \text{leaveGroup()} \), and \( \text{joinGroup()} \), to remove a member and to add a member, respectively.

```java
class Group {
    private Vector G;
    private String groupname;
    public Group (String groupname) {} 
    public void leaveGroup (String member) {} 
    public void joinGroup (String member){}
}
```

Fig. 3. The system architecture.
- class Message. This object forms messages used in the system by keeping a local counter for attaching a sequence number to each multicast message.

- class ConnToMembers. This object sets up the stream-based connection to each member in the group. Streams are collected into a vector, channels, which is used for multicasting.

```java
class ConnToMembers { // connecting to other members.
    private Group G;
    private Vector channels;
    public ConnToMembers(Group G, Vector channels) {}
    public void mCast(Message m) {}
    public Message readMcast() {}
}
```

- class Deliver. This object delivers requests being added to inQ. In the normal execution (not performing any protocol), messages are delivered in FIFO ordering.

```java
class Deliver {
    private Group G;
    private Vector channels;
    private Vector inQ = new Vector();
    private Vector outQ = new Vector();
    public Deliver(Group G, Vector channels) {
    }
    public void deliverReq() {}
    public Message deliverFlush() {}
    // deliver only m(h).
    public Message deliverCombined() {}
    // deliver only m(r).
}
```

- class Flush and class McastAtomAndFlush. A Flush object is used to perform the flush protocol. A McastAtomAndFlush object performs the multicasting atomicity protocol and the flush protocol.

```java
class Flush {
    private char[] flush_array;
    public Flush (Vector channels, ConnToMembers conn) {}
    public void flushing(Message m) {}
}
class McastAtomAndFlush {
    private char[] flush_array;
    public McastAtomAndFlush(Vector channels, ConnToMembers conn) {
    }
    public void flushing(Message m) {}
}
```

- thread classes. An AcceptClients object is a thread for accepting client connections. When there is a connection from a client, it spawns a child thread to that client which is a thread of AcceptClient. A McastReading is a thread of reading all multicasts from peer members so that the main thread can be free from this matter to concentrate on request handling.

```java
class AcceptClients extends thread {}
class AcceptClient extends thread {}
class McastReading extends thread {}
```

- class Surrogate. An instance of surrogate is a replica which runs on a workstation. A surrogate object is the control object of all other objects.

```java
class Surrogate {
    private Group G;
    private Vector channels = new Vector();
    private ServerImpl service = new ServerImpl();
    private ConnToMembers conn;
    private Deliver deliver;
    private Flush flusher;
    private McastAtomAndFlush failHandler;
    public void Surrogate (Group G) {
        conn = new ConnToMembers(G, channels);
        deliver = new Deliver(G, channels);
        flusher = new Flush(channels, conn);
        failHandler = new McastAtomAndFlush (channels, conn);
        ServerSocket server = new ServerSocket();
        AcceptClients accept_clients = new AcceptClients(server);
        accept_clients.start();
        McastReading read_mcasts = new McastReading(conn);
        read_mcasts.start();
    }
    public void handleUpdateReq (Message m) {
        conn.mCast(m);
        service.upCall(m);
    }
    public void requestHandler() {
        while (true) {
            Message m = deliver.deliverReq();
            if m is m(q) then service.upCall(m);
            if m is m(u) then handleUpdateReq(m);
            if m is m(p) then service.upCall(m);
            if m is m(j) then flusher.flushing(m);
            if m is m(l) then flusher.flushing(m);
            if m is m(h) then flusher.flushing(m);
            if m is m(t) then failHandler.flushing(m);
            if m is m(r) then failHandler.flushing(m);
        }
    }
}
public static void main() { // the coordinator or a non-coordinator member.
    Group G = new Group();
    Surrogate replica = new Surrogate(G);
    replica.requestHandler();
}

5. The constructing tool

In essence, the tool for constructing a replicated service system based on the active replication scheme and the object-oriented design pattern includes three components: the replica group interface definition language (RGIDL), the RGIDL preprocessor and a runtime system.

5.1. The RGIDL language

The RGIDL language is used to specify a replica group. The declaration about a replica group includes group name, group style, initial members (if there is any), update ordering constraints, and a service interface embracing a set of operations exported to clients. Table 1 shows the RGIDL syntax in EBNF.

A replication system is defined by two parts: a group definition part and a service interface definition part.

- **The group part.** It gives the following needed description about a replica group.
  1. **Group name.** The group name is a unique identifier to be referenced by clients and registered to NS.
  2. **Group style.** Two styles, PP and PB, are used to specify a primary-peer (PP, i.e. active replication) group and a primary-backup (PB) group, respectively.

- **MemberList.** It gives a list of initial members on which the group will be running, and the group port number at which clients can make connections. Initial members can be an empty set, in this case, the primary is started first by registering itself to the NS, other members then join the group one by one.

- **NSRef.** The host name and port number of the NS.

- **Propagation.** It defines the propagation frequency in two ways: (1) Every number of requests, e.g. every 10. If the frequency is defined as every 0, it implies there is no propagation among replicas. This can be used for services that only contain query type of operations. Whereas if it is defined as every 1, it implies every update is sent to other replicas right away. (2) A time interval, e.g. every 6 ms, where a timer is setup to trigger the propagation every 6 ms.

- **The service interface part.** It defines a set of operations that the server exports to clients. OperationDef defines an operation with two types of specifiers: (1) operation type: query or update, (2) ordering constraint. If an operation is defined as an update operation, an ordering constraint can be added to it. The types of ordering constraints are: causal, total and atocausal (stands for total + causal). Ordering constraints are only applicable to PP groups, where FIFO is the default ordering constraint. The rest of the operation is defined by JavaMethodDefinition which follows the same syntax of any Java method definition.

### Table 1: The RGIDL syntax in EBNF

<table>
<thead>
<tr>
<th>Syntax</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>`ReplicaDef ::= 'replica' Ident '}' GroupDef {InterfaceDef</td>
<td>}`</td>
</tr>
<tr>
<td>`GroupDef ::= 'group' Ident 'PP'</td>
<td>'PB'</td>
</tr>
<tr>
<td><code>GroupConfig ::= MemberList NSRef Propagation</code></td>
<td>Contains information about members, NS, and propagation.</td>
</tr>
<tr>
<td>`MemberList ::= 'site' '+' [IPNameList] '+'</td>
<td>PortNum `</td>
</tr>
<tr>
<td><code>IPNameList ::= IPName ',' * IPName</code></td>
<td>Specifies the list of IP names.</td>
</tr>
<tr>
<td><code>NSRef ::= 'NS' ':' IPName '.' PortNum</code></td>
<td>Represents the NS and its port number.</td>
</tr>
<tr>
<td><code>Propagation ::= 'propagation' ':' every Number [msg]</code></td>
<td>Describes the propagation pattern.</td>
</tr>
<tr>
<td><code>PortNum ::= Number</code></td>
<td>Indicates the port number.</td>
</tr>
<tr>
<td>`InterfaceDef ::= 'interface' Ident '}' (OperationDef</td>
<td>)`</td>
</tr>
<tr>
<td><code>OperationDef ::= [OpType] [OrderConstraint] JavaMethodDefinition</code></td>
<td>Specifies an operation.</td>
</tr>
<tr>
<td>`OpType ::= 'query'</td>
<td>'update'`</td>
</tr>
<tr>
<td>`OrderConstraint ::= 'total'</td>
<td>'causal'</td>
</tr>
<tr>
<td>`IdentList ::= (Ident</td>
<td>'}'</td>
</tr>
<tr>
<td><code>Ident ::= [a-zA-Z]+ [0-9]*</code></td>
<td>Identifiers.</td>
</tr>
<tr>
<td><code>Number ::= (0-9)</code></td>
<td>Integers.</td>
</tr>
<tr>
<td>`IPName ::= [Ident</td>
<td>']'</td>
</tr>
</tbody>
</table>

5.2. The preprocessor and runtime system

With a given RGIDL specification file, the preprocessor generates three sets of files: (1) the service interface file containing a set of Java methods (operations exported to clients); (2) client driver (client object) and client proxy files to form the client program; and (3) server driver (service implementation object) and server surrogate files to form the replica program. Fig. 4 depicts this procedure.

The runtime system supports the execution of both client programs and replica programs. The runtime system helps achieving server distribution and replication transparency. The design principle for the runtime system is to layer objects so that high layer objects are based on and supported by the lower layer objects. The runtime system is developed on the basis of Java networking package and itself is a package consisting of a collection of objects: utility objects, mini-protocol objects, root proxy and root surrogate. Fig. 5 depicts the layers of the runtime object model.
surrogate object to be inherited by all particular replica surrogates. The root surrogate is designed by employing the following ideas:

1. Communications are based on reliable TCP/IP channels.

2. Multithreading is used to improve the performance of the surrogate. Clients can make connections at any time, each client is handled by a separate thread which is spawned at the connection time to receive and reply the client requests. Communications among replicas of the group are handled by a separate thread and a few other threads are established to handle concurrent matters in the surrogate.

3. The surrogate object is a composite/aggregate of mini-protocol objects and utility objects.

6. Performance evaluation

6.1. Metrics

The evaluation is based on two metrics: the average response time (ART) over requests, and the average system throughput (AST) which is the average number of requests that can be executed by the system.

Assume we have an n-replica system; $R^r$, where $R^r = \{R_1, \ldots, R_n\}$. To evaluate the overall ART of the replication system, assuming the ART at each replica is $ART_{R_i}$, where $1 \leq i \leq n$, then we define the overall $ART_{R^r}$ to be the average value of average response times achieved at each replica:

$$ART_{R^r} = \frac{1}{n} \sum_{i=1}^{n} ART_{R_i}.$$  

(1)

To evaluate the overall AST, we assume that $N$ requests are received by each replica of the group. The replicated system starts from a global consistent state. By executing all $n * N$ at all replicas, the replicated service system stops at another globally consistent state. Let $AST_{R_i}$ represent the AST at $R_i$; it is measured by the average number of requests executed at $R_i$ to finish all $n * N$ requests, then we can define the overall $AST_{R^r}$ to the minimum value among all $AST$s achieved by each replica:

$$AST_{R^r} = \min(AST_{R_1}, \ldots, AST_{R_n}).$$  

(2)

That is, the overall system throughput is determined by the replica who is the last one to finish executing all $n * N$ requests.

However, these two metrics can be largely affected by many factors, including the following:

- Replication degree. This relates to the number of replicas employed in the system. The response time over queries can be greatly improved if more replicas are involved. For example, eight clients connect to two
different replica system settings: (1) two replicas; (2) four replicas. With the first setting, each replica is connected by four clients. With the second setting, each replica is connected by two clients. If all requests are queries, the response time in the second setting can be faster than in the first one, since all queries are handled by local replicas without communicating with peer replicas. In contrast to a non-replicated system, the system throughput of a four-replica system is four times over the non-replicated system, if all requests are queries.

However, the response time over update operations is complicated. The response time also depends on other factors, such as the update percentage rate and the strength level of ordering constraints being placed on the set of update operations. One thing we can be sure is that, the more replicas involved, the more network communication traffic is generated.

- **The number of clients connected to the replicated system.** In general, the more clients connected to the system at the same time, the slower the response time is. However, if each client only sends requests in a very slow pace, i.e., the request arrival rate is very low, then the response time should not be significantly affected.

- **Update percentage.** Service replication systems are very useful and efficient in an application environment where the percentage of update operations is low. If the percentage of updates is high, the performance of the replicated system can be worse than that of the non-replicated system. For instance, if the update percentage is 100%, and all updates are to be totally ordered, the replicated system certainly performs worse than the non-replicated system. The reason is that all updates have to be executed in the same order at all replicas (no concurrency at all), plus each update involves network communications and the network delay. Whereas in the non-replicated system, no network communication is needed. Thus, the higher update percentage rate is, the less efficient (both the response time and system throughput) the replicated system becomes.

- **Propagation frequency.** If the propagation happens more often, then the size of the propagation message is small. The response time is affected by relatively small delays at short time intervals, and the delay is the result of sending out the propagation message by the underlying communication primitive. When the propagation happens less frequently, the propagation message grows large. The response time is affected by relatively large delays but at longer time intervals. Thus, the ART is determined by whoever becomes the prominent factor—the more frequent small delays or less frequent large delays.

In this paper, we concentrate on the study of two metrics, the ART and the AST, with respect to propagation frequencies, but fixing the replication degree to four replicas and update percentage to 100%.

Instead of calculating the ART and AST, we calculate \(\text{ART}_1, ..., \text{ART}_n\) and \(\text{AST}_1, ..., \text{AST}_n\) to show the detailed testing results on different machines.

### 6.2. The system setting

In our experiments, we allocated four Sun Sparc stations for running four replicas. These four Sun stations have slightly different hardware configurations. Each replica is fully connected with other three replicas by TCP/IP reliable streams. When a replica is first started, according to its ranking (a pre-established order which is the order placed in the group membership list), it either awaits the connections from peer replicas or makes connections to peer replicas. After connections are created, all child threads of each replica are started. These threads compete for the CPU time. Sending propagations and receiving propagations are handled by using two separate child threads in parallel.

We use a service application which is composed of eight different update operations, represented by index numbers from 0 through 7 inclusive. Each replica has a child thread simulating a client that issues requests one after another, in other words, after issuing a request, the client thread is blocked and awaits the reply from the replica (which is the main thread), after receiving the reply, it then issues another request. In the following experiments, each client thread sends a sequence of 100 requests, which is generated by a random function that produces well-balanced numbers (between 0 and 7 inclusive) to simulate the sequence of operations in the real environment.

Thus we have \(n = 4, N = 100\). The ART at each replica is evaluated by the average response time over 100 requests in millisecond. The AST is measured by the average number of requests executed per second at each replica by finishing 400 requests (100 requests are issued by its client, and 300 are propagated from peer replicas).

Because we use a client thread within the replica to simulate the client, if the average network delay of a round trip (sending a request and receiving the reply) for a request is \(\delta\), then the real ART should be \(\text{ART} + \delta\).

All our experiments are carried out in an environment composed of Sun SPARC stations connected by a local 10 Mbps Ethernet. The programming environment is under Java 1.2 (Jaworski, 1998) and Sun Solaris 5.6. The most involved Java packages are Java network (java.net) and Java input/output (java.io) packages (Arnold and Gosling, 1996; Harold, 1997; Sridharan, 1997).
6.3. The effect of propagation frequency

This experiment is to study how the propagation frequency affects the response time and the system performance. Propagation frequency affects a system where most of operations are commutative or causal. This is because if an operation is a total or total+causal operation, the propagation is triggered right away. In other words, total or total+causal operations break up the regularity of propagations at the specified frequency.

In this particular experiment, we assume that all eight operations are commutative operations so that the propagations happen exactly at the frequency specified. We let the propagation frequency vary at: (1) every 1 request; (2) every 2 requests; (3) every 5 requests; (4) every 10 requests; (5) every 20 requests; (6) every 50 requests.

The test also employed asynchronous propagation method, which means a request is handled and replied to the client right away. In contrast, the synchronous propagation requires a propagation to reach peer replicas before the request is handled and returned to client. Asynchronous propagation normally gives a quicker response time.

Four Sun stations involved in this test are: "bofur", "bifur", "dwar" and "elving". Since each of them has a slightly different hardware configuration from others, in turn each machine has shown a slightly different performance.

Fig. 6 shows the testing results of ARTs over 100 requests at varying propagation frequency rates on four machines. From the figure, we can observe that at the frequency of every 5 and 10 requests, the ARTs are at the lowest level (the best). When propagation happens more frequently, i.e. less than every 5 requests, the ARTs tend to be higher. When the propagation happens less frequently, i.e. every more than 10 requests, the ARTs tend to grow slightly. This can be explained by the fact that the ART is determined by whoever becomes the prominent factor, either the more frequent small delays or less frequent large delays, as we discussed in Section 6.1 under propagation frequency.

Fig. 7 shows what the ASTs are achieved on four machines. At the frequency of every 5 and 10 requests, replicas have relatively high throughput rates. This matches the test results depicted by Fig. 6, where the ARTs at the frequency of 5 or 10 are the lowest, in turn, highest throughput rates should be achieved there.

In the implementation, delivering client requests and delivering received propagations are handled by two separate child threads in parallel as well. Threads can be set to have a highflow priority. The experiment shown in Fig. 8 is organised by setting the thread of delivering client requests to have a higher priority than that of delivering received propagations.

We also conducted an experiment by giving all threads an equal priority, the result is shown in Fig. 7. The ARTs are affected by this setting quite significantly.

![The impact to ARTs](image1)

**Fig. 6. The ARTs at different propagation frequencies.**

![The impact to ARTs](image2)

**Fig. 7. The ASTs at different propagation frequencies.**

![The impact to ARTs](image3)

**Fig. 8. The ARTs when threads are assigned different priorities.**
Also we can observe that the lowest ARTs are shifted to frequencies at 10 and 20. Fig. 9 shows the comparison in ARTs of these two experiments on the same machine "durin". The top curve represents the test result of assigning all threads an equal priority, whereas the bottom curve represents the test result of assigning the delivering-client-request thread a higher priority. The results demonstrate, by giving client requests a high priority, the ART is shortened quite dramatically.

7. Remarks

The design and implementation of an active replication scheme, a design pattern based on the scheme, and a constructing tool for developing distributed object replications in COWs have been presented. The pattern separates a replicated and distributed object into two interrelated components, the service implementation object (the service object) and the replication management object (the surrogate). An interface between these two objects is through the public interface ServiceInterface containing a method upcall(). The surrogate object manages the connections with other surrogates and clients; it intercepts clients' requests, propagates update requests to other surrogates, and issues up-calls to the service object; it executes protocols in the event of a member's join/leave/fail. To facilitate the design of the active replication model, two group communication services, multicasting atomicity and membership atomicity, are embedded in the surrogate object. The atomic multicasting service guarantees members receiving the same history of updates. Whereas the dynamic membership service guarantees that membership changes do not bring inconsistency to the replication system. The pattern is general enough to the design of any object replication system. The constructing tool allows the developer of a replicated service system to concentrate on the implementation of service programs and client interface programs while leaving the replication control protocol to be generated by the tool. This can shorten the development life-cycle and reduce the complexity significantly.

Acknowledgements

This research was partially sponsored by the following grants: (1) Australian Research Council Small Grant No. 0504.23409.0132.35.01, (2) Strategic Grant, City University of Hong Kong under No. 7001060, and (3) RGC Hong Kong grant under No. 9040352.

References


Li Wang is a senior software developer at Phoneware Communication Systems Pty Ltd, Australia. She received a Ph.D. in Computing and Information Systems from Deakin University, Australia. Her Ph.D. focused on constructing distributed service replication systems by support of toolkit. Her special interests include object-oriented design methodologies for distributed systems, and network communication protocols design, implementation and evaluation. Contact her at Li_Wang@phoneware.com.au.
Wanlei Zhou is currently an Associate Professor and the Deputy Head of School of Computing and Mathematics, Deakin University, Melbourne, Victoria, Australia. Wanlei received the B.E. and M.E. degrees from Harbin Institute of Technology, Harbin, China in 1982 and 1984, respectively, and the Ph.D. degree from The Australian National University, Canberra, Australia, in 1991. Before joining Deakin University, Dr. Zhou has been a programmer in Apollo/HP at Massachusetts, USA, a Lecturer in National University of Singapore, Singapore, and a Lecturer in Monash University, Melbourne, Australia. His research interests include distributed computing, computer networks, databases, performance evaluation, and fault-tolerant computing. Dr. Zhou is a member of the IEEE and IEEE Computer Society. Dr. Zhou can be contacted via wanlei@deakin.edu.au

Weijia Jia received his B.Sc., M.Sc., and Ph.D. in computer science. He is currently an associate professor of Department of Computer Science, City University of Hong Kong. His research interests include network, parallel and distributed systems, multicast protocols, real-time communications, and Internet routing. He has published over 100 papers in journals and conferences. He served as PC members and Program Co-Chairs for various International Conferences. He is a member of IEEE Computer Society. Dr. Jia can be contacted via wjia@cs.cityu.edu.hk.
THE DESIGN OF A MOBILE DECISION SUPPORT SYSTEM

YUE WU1, TIAN YONGHONG1, AND WANLEI ZHOU2

1College of Computer Science and Engineering, The University of Electronic Science and Technology, Chengdu, China, 610054. Email: ywu@uestc.edu.cn

2School of Computing and Mathematics, Deakin University, Melbourne, VIC 3168 Australia. Email: wanlei@deakin.edu.au

The emergence of mobile computing environments brings out various changes in the requirements and applications involving distributed data and has made the traditional Intelligent Decision Support System (IDSS) architectures based on the client/server model ineffective in mobile computing environments. This paper discusses the deficiencies of the current IDSS architectures based on data warehouse, on-line analysis processing (OLAP), model base (MB) and knowledge base (KB) technologies. By adopting the agent technology, the paper extends the IDSS system architecture to the Mobile Decision Support System (MDSS) architecture. The logical structure and the application architecture of the MDSS and the mechanisms and implementation strategies of the User Access Agent System, a major component of the MDSS, are described in this paper.

Keywords: IDSS, mobile agent, MDSS.

1 Introduction

The development of Data Warehouse (DW) and On-Line Analysis Processing (OLAP) technologies has opened a new dimension for building Decision Support Systems (DSS) [1]. The Intelligent Decision Support System (IDSS) that combines the technologies of DW, OLAP, Model Base (MB), knowledge base (KB), and data mining is a higher-level decision support system [13]. The architecture of an IDSS can be depicted in Figure 1.

In the Client layer, clients interact with the User Interface of the IDSS, which in turn interacts with the middle layer of the application servers, such as OLAP and data mining. These middle layer servers will interact with the lower layer data servers, such as the data warehouse, the knowledge base and the model base.

Although IDSS can improve the speed and flexibility of decision-making by integrating large amount of decision-making data and information in the data warehouse, it has the following deficiencies:

1. Data in the data warehouse are synthesized and stored according to the decision-making subjects and they are relatively stable. So the IDSS cannot provide a real-time integration of decision-making data for distributed decision-making.

2. IDSS cannot provide transparent accesses required by the personalized decision-making information. In an IDSS, the contents and formats of the user
sharable (or accessible) decision-making information are determined by the OLAP or the data mining servers. Users have to understand the abilities and access methods of these analysis applications.

3. IDSS does not support the mobile computing mode. The emergence of the mobile computing environments has changed the ways we process and utilize distributed data. Compared to traditional distributed computing environments in fixed networks, mobile computing environments have properties such as mobility (including mobility of computing platforms, mobility of access resources, and personal mobility), and network adaptability (including frequent network disconnections, low reliability, asymmetrical network communications). The IDSS based on the traditional client/server model can no longer satisfy the requirements of mobile computing environments. Therefore it is necessary to introduce a new and flexible architecture.

![Diagram of IDSS application architecture](image)

**Figure 1. IDSS application architecture**

In recent years, the agent technology has attracted attentions from many researchers. The agent technology, in particular the mobile agent technology, provides a new approach for the analysis and design of distributed and open systems in mobile computing environments. In this paper we adopt the mobile agent technology in distributed decision support systems and extend the IDSS to a mobile decision support system (MDSS).

2 **Mobile Agent Technology**

The concept of agents first appeared in the research of distributed artificial intelligence, in which they are defined as programs or computing entities used to model the behaviors of human beings and to provide related services, whereas mobile agents are those agents who are dynamic and distributed. That is, mobile agents are autonomous software entities with some intelligence. They can autonomously represent the user or other software entities to move among network
nodes in order to search for some resources or to complete some specific functions. Because of the introduction of distributed mobile computing concepts, the mobile agent technology can successfully deal with tasks that cannot be performed well by the traditional client/server model. Agent technology has the following advantages and characteristics:

- Dynamic execution.
- Asynchronous computing.
- Parallel computation.
- Intelligent routing.

The characteristics and advantages of the mobile agent technology enable mobile agent systems to work smoothly in an environment of low bandwidth and unstable network connections, and therefore are suitable to mobile users. Introducing the mobile agent technique into the mobile computing environment, especially the mobile decision support systems, will bring a number of advantages, for example, it can support mobile users to carry out decision analysis in an unstable network connection situation, and can reduce the information transmission dramatically. Mobile agents can provide mobile users with personalized services, and can autonomously perform the induction according to the knowledge of network capacity and task models, relieving mobile users from knowing the capacities and access methods of individual decision support systems.

3 The MDSS Model Based on the Agent Technology

3.1 The Logical Structure of the MDSS

We can abstract the IDSS logical structures based on the traditional client/server model into a two-layered structure consisting of a Source Schema and a Client Schema. The source schema is also called the server schema, and in a client/server system using the "lean client machine" and the application server structure the server schema can be further categorized into two sub-schemas: an Application Schema and a Data Schema. The data schema describes the logical structure of the global data in the IDSS data warehouse; it is the global conceptual view of the IDSS. The application schema describes the logical structure of the analysis data of DSS applications (such as OLAP and DM applications) within the IDSS; it is the complete set of the decision support information schemas provided by the IDSS for users. The client schema is the sub-set (can be the complete set in special cases) of the application schema; it is actually the decision support information schemas presented to users.
Our MDSS structure adds a new layer, the *agent layer*, between the client and server of the IDSS. That is, it has a three-layered structure consisting of the source schema, the agent schema, and the client schema. In the MDSS, the source schema describes the global data, including data sources from the data warehouse, the OLTP database, and the external data, and the logical structure of the analysis data of DSS applications on top of the global data. The data warehouse is the main data source for the MDSS to perform its decision analysis and processing. The OLAP and data mining based on the data warehouse carry out the main tasks of information analysis. In order to be logically symmetric, we use the term of *information sources* to represent the data warehouse, the OLTP database, and the external data. Figure 2 compares the logical structures of the IDSS and the MDSS.

![Diagram](image_url)

**a) IDSS logical structure.**  **b) MDSS logical structure**

*Figure 2. Comparison of logical structures*

In the MDSS, the function of the agent is to access various data sources and DSS applications on behalf of the user. That is, it acts like an analyst. Therefore, we represent the agent layer as the agent schema in Figure 2. The agent schema is the bridge between the application schema and the client schema; it describes a special client schema, and is the conceptual view of the application schema.

### 3.2 The Application Structure of the MDSS

Figure 3 depicts the four-layered system application structure of the MDSS based on the agent technology. The application structure of the MDSS consists of the following components:

1. **Data platform of the MDSS**: Data platform is the DSS data sources used for the organization of analysis data. MDSS data platform includes data warehouse, the OLTP database, the external database, and other data sources, such as file system, electronic forms, E-Mails, etc. The fundamental data platform of the MDSS is data warehouse. The data warehouse integrates the original operational data decentralized in an enterprise and data received externally, and carries out most of the decision analysis.

2. **Data analysis and application of the MDSS**: In the application structure of the MDSS, all DSS applications are provided to the User. Access Agent System...
(UAAS) in terms of DSS service methods. DSS service methods are relatively stable in comparison with other DSS application programs. That is, in certain time interval, the interface of a DSS service method remains unchanged. The basic data analysis of the MDSS is OLAP. Based on multi-dimensional analysis, the OLTP describes the requirements for the multi-layered, multiangled analysis and processing of data during the management and decision-making processes. In the mean time, data mining and traditional DSS analysis are also used as the means for complementary analysis.

![Diagram of Application structure of the MDSS](image)

**Figure 3. Application structure of the MDSS**

3. Knowledge base and model base of the MDSS: The domain base plays a crucial role in the MDSS. When an agent carries out the process of decision analysis based on multiple information sources, it needs to use knowledge from the domain knowledge base during every step of the process. Domain base is an extended model base and method base; its contents include the global information search schema, the logical structure of the information source and stored information, the semantics of decision search and security authentication information, interfaces of the application methods for various DSS decision information applications, decision analysis rules for various information sources, etc.

4. Middleware of the MDSS: The middleware of the MDSS is the User Access Agent System. It accepts decision analysis requests from mobile users through friendly interfaces; migrates among various DSS application systems according
to certain strategies; and invoking various DSS application methods to carry out decision analysis in order to obtain the information required by the user.

Compared to the IDSS in Figure 1, the MDSS has the following characteristics:

- **Distribution and cooperation**: MDSS is an extension of the distributed decision support system. It also requires various decision support applications to cooperate via the agent to accomplish the decision analysis.

- **Mobility**: The UAAS allows mobile users to access the decision support systems in the situations where the network connections are unstable and access positions are mobile.

- **Autonomy**: Through the UAAS, the decision support applications of the OLTP system based on data warehouse and other data sources can be independently designed, constructed, and managed.

- **Adaptability and scalability**: Various decision support application systems, no matter it is the OLTP based on data warehouse, the data mining, or the DSS applications based on the OLTP data sources, can be smoothly integrated with users through UAAS.

![Diagram of UAAS](Image)

**Figure 4. The structure of the UAAS**

4 **The User Access Agent System**

The kernel of the MDSS application structure is the User Access Agent System (UAAS), depicted in Figure 4. The UAAS includes a User Agent Subsystem (UAS), an Agent Server Subsystem (ASS), and an Application Agent Subsystem (AAS).
• The UAS provides users with personalized interfaces and interacts with the user. It accepts the decision analysis requests from users and passes the requests to the ASS. The UAS consists of a User Agent (UA) and a User-access Domain Base (UDB).

• The ASS decomposes and maps the decision analysis requests from users, coordinates the decision analysis in the system on behalf of the users, and returns the results in appropriate format back to the user agent. The ASS includes the Analysis Agent (AA), Domain Agent (DA), Agent Manager (AM), Access Cache (AC) and Global Domain Base (GDB).

• The AAS is consisted of the Knowledge Agent (KA) based on the DSS data sources, the Agent Facilitator (AF), and the Local Domain Base (LDB). It is responsible for the interaction between the local DSS applications and data sources. It also provides decision analysis and processing functions for certain (e.g. local) decision supporting information.

Note that all agents in the UAAS are static agents except for the analysis agent (AA). According to the decision analysis requests from users, the AA migrates within the system and coordinates various DSS applications to perform a coordinated decision analysis to obtain the information required by users.

4.1 Description of the UAAS Components

4.1.1 Description of the Agents

An Agent in the UAAS is an active object \(^{[12]}\) of \((D, M, SD, P)\), in which \(D\) is a set of local data, \(M\) is a set of methods, \(SD\) is a set of structural dependency relationship among the methods (including trigger dependency, order dependency, real-time dependency, etc.), and \(P\) is a set of interruption and re-synchronization points.

The functions of each Agent in the UAAS can be described as follows:

• The User Agent (UA) is the interface between the UAAS and the user. It includes the following functions: (1) user login and authentication; (2) deciding the user accessible decision information resources and access interfaces, and creating personalized access interfaces according to the user requests; (3) accepting and formalizing user requests for decision analysis, and presenting these requests to the analysis agent (AA); (4) receiving the analyzing results from the AA and returning them to the user; (5) copying the knowledge required by the users from the global knowledge base when the user agent subsystem is started.

• The Analysis Agent (AA) is used to process the user requests for decision analysis. It needs to inquire from the domain agent (DA) about the processing methods for the user decision analysis requests and the corresponding domain schemas. It then uses the divide-and-conquer strategy to solve the global
analysis requests from the users, and creates sub-analysis agents to execute the
tasks of decision analysis. The sub-analysis agents can use remote processing or
migration to interact with the knowledge agent of the service agent sub-system.

- The Domain Agent (DA) dictates the domain schemas that satisfy the user
analysis requests through the access to the global domain base (GDB). It also
collaborates with the knowledge agent to maintain the data consistency between
the local domain base and the global domain base, and collaborates with the UA
to maintain the data consistency between the user access domain base and the
global domain base.

- The Knowledge Agent (KA) has the following three functions: (1) invoking the
DSS application services to carry out the decision analysis on the local
information sources through the original database methods and application
methods; (2) learning of the local domain knowledge; and (3) maintaining the
data consistency between the local domain base and the global domain base
through write-through or write-back strategies.

- The Agent Facilitator (AF) is used to organize and manage the receiving,
servicing, and starting of the AA when it migrates to the DSS application
server, and to interacting among AAs.

- The Agent Manager (AM) is the manager of the UAAS and has the following
functions: (1) the management of the security and authentication of the UAAS;
(2) the service for the life cycle of agents in the UAAS, which includes the tasks
for agent creation, execution, suspension, recovery, and termination; (3) the
resource distribution of UAAS, including the distribution of memory and file
resources; (4) position service for mobile agents, and following the positions of
the frequently migrating agents in the UAAS; and (5) the management of an
access cache to preserve the analysis results during network disconnections.

4.1.2 Management of the Access Cache

In order to support the access of decision support systems by mobile user machines
in the case of unstable network connections, the MDSS adopts the access cache
technique. That is, a data cache, managed by the AM, is attached to the agent server
to temporarily store the decision analysis results of mobile users. By using this
mechanism, a mobile user machine can have an unstable connection with the MDSS
during the period when the UAAS is processing its decision analysis request. When
the UAAS completes the user's analysis request, it automatically stores the analysis
results into the access cache. The AM will automatically return the results to the
mobile user machine according to its present position when the mobile user machine
logs into the MDSS again.
4.1.3 The Organization of the Domain Base for Supporting Mobile Computing

In order to support mobile computing, the MDSS organizes its domain bases in a layered architecture. A layered architecture can improve the security of the system and balance the load to the domain bases, therefore reduce the information traffic during the decision analysis and support the decision analysis search effectively in an unstable network connection situation. The organization of the MDSS domain bases is depicted in Figure 5. It has the following characteristics:

- The domain base of the MDSS is divided into three layers: the local conceptual layer, the global conceptual layer and the user view layer. These layers correspond to the local domain bases, the global domain base, and the user-access domain bases (UDB). The domain bases in these three layers form a unified distributed domain base system. In this system, all local domain bases distributed on DSS application servers in various information sources form the local conceptual layer. They store the local information for local DSS information sources. The global domain base is stored in the server forming the global conceptual layer. The user-access domain base is the decision support schema that a current user can access and is a sub-set of the global domain base. The UDB also stores the information of the personalized decision access schema for the user.

- Strategies for data consistency: The data consistency in the domain bases has two aspects: the first is the knowledge consistency between the global domain base and the local domain bases, and the second is the knowledge consistency between the global domain base and the user-access domain bases.

![Figure 5. Organization of the MDSS domain bases](image)

The sources that cause the inconsistency between the GDB and the LDBs include: updating the data of the local data source causes the change of the data analysis schema; changing of the local DSS application methods; stopping the use of a local DSS application system because of certain reasons, etc. A monitoring protocol can
be used to maintain the knowledge consistency between the GDB and LDBs. The concept of the monitoring protocol was originally used in the consistency maintenance for caches of multiple processors in parallel processing systems [4]. In the design of the MDSS, knowledge agents can monitor the changes of the analysis schema in local DSS. When these changes cause a change of the system analysis schema, the KA notifies the DA to invalidate the changed analyzing subject, or to carry out an updated processing. This method is the monitoring protocol. There are two strategies for maintaining the knowledge consistency between the GDB and the LDBs: via write-through or via write-back. Because the amount of data used in maintaining the knowledge consistency between the GDB and the LDBs is relatively small, using the write-back strategy will not form a processing bottleneck when the system runs smoothly. So we adopt the write-back method in our MDSS.

The sources that cause the inconsistency between the GDB and the UDBs include: the change of the analysis schema in the GDB causes the invalidation of the existing analysis schemas of the UDB; the user uses an unregistered user machine to access the MDSS, etc. The maintenance of knowledge consistency between the GDB and the UDBs is relatively straightforward. When an analysis schema of the GDB changes, the UAAS records the contents of the change. When a user logs into the system and the UAAS discovers that there exists changed analysis schema, the changed information will be passed to the mobile user machine to update the UDB. When a user accesses the MDSS using an unregistered user machine, after passed through the authentication procedure, a new UDB is created for the user machine and the information of the user accessible analysis schema is copied to the user machine.

4.2 The Workflow of the UAAS

The workflow of the UAAS can be described in the following:

1. A user registers and invokes the local user agent subsystem (UAS). If the user is the first time to register or the access information is to be changed, the UAS copies the accessible knowledge of the user from the GDB.

2. The user agent (UA) accesses the GDB; authentication of the user and the privileges of the user; the user inputs the decision analysis request via an access interface; the UA passes the request to the analysis agent (AA).

3. The AA inquires the domain schema for the decision analysis request from the domain agent (DA)

4. The DA returns the matched information to the AA; the AA decomposes the user's global request and decides the following processing methods according to the result of the decomposition.

- If the global request only involves one DSS application, then the AA uses the remote processing method to access the knowledge agent (KA) of the DSS application server.
• If the global request involves multiple DSS applications, then the AA spans a number of sub-analysis agents (SAA). These SAAs use the migration method to access the KAs on the related DSS application servers. The AA then becomes an agent coordinator to coordinate the communication and collaborated analysis of the SAAs to complete the user's global analysis request.

5. The SAA that has migrated to the local DSS server asks the local KA to process the local analysis request; the KA first carries out the mapping of the local analysis request, producing the execution sequence of local DSS analysis methods, then the KA invokes the DSS service methods to carry out the decision analysis on the information sources in the local DSS to obtain the data or decision information required to satisfy the user decision analysis. During the process of analysis, the SAA constantly carries out collaborated operations according to the commands of the AA and communicates with other SAAs. After the completion of local analysis, according to the analysis control workflow, the SAA migrates to a next DSS server or returns to the AA.

6. The AA integrates all analysis data or decision information and stores the results in the access cache of the agent server; the agent manager (AM) returns the analysis results to the proper user agent (UA) according to the position information of the mobile user found from the position service of the AM.

7. The UA returns the analysis results to the user and continues to process user requests or goes into sleep.

5 Remarks

The widely spread use of mobile computers requires us to provide effective means for mobile users to access enterprise information systems and the technology of decision support has become a crucial factor of information strategies of modern enterprises. However, how to construct a flexible and effective decision support system on the distributed information systems of enterprise, in order to support mobile users to access the decision support system in any time and from anywhere is still unclear. The model for constructing mobile decision support systems based on the agent technology presented in this paper provides a way to solve this problem.

We are currently building a prototype based on our model and are carrying out further researches on the model, including the learning issues of styles and interests of the user-access system, the security issues of the MDSS in a mobile computing environment, the performance issues, and so on.

In the MDSS, a user request of decision analysis (called the global analysis request) involves multiple data sources and DSS analysis applications. How to quickly and effectively decompose the global analysis requests into sub-analysis requests for various analysis sources is crucial to the performance of the whole system. It is therefore necessary to find an effective strategy to decompose a global
analysis request into sub-analysis requests for individual analysis sources. We discuss the decision analysis techniques based on multiple analysis sources for the MDSS in another paper.

References

THE DEVELOPMENT OF A MOBILE DECISION SUPPORT SYSTEM

WU YEU, TIAN YONGHONG

College of Computer Science and Engineering, The University of Electronic Science and Technology, Chengdu, China, 610054. Email: ywu@uestc.edu.cn

WANLEI ZHOU

School of Computing and Mathematics, Deakin University, Melbourne, VIC 3168 Australia. Email: wanlei@deakin.edu.au

Received: July 7, 2000. Revised: February 20, 2001. Accepted: May 8, 2001

Abstract

The emergence of mobile computing environments brings out various changes in the requirements and applications involving distributed data and has made the traditional Intelligent Decision Support System (IDSS) architectures based on the client/server model ineffective in mobile computing environments. This paper discusses the deficiencies of the current IDSS architectures based on data warehouse, on-line analysis processing (OLAP), model base (MB) and knowledge base (KB) technologies. By adopting the agent technology, the paper extends the IDSS system architecture to the Mobile Decision Support System (MDSS) architecture. The logical structure and the application architecture of the MDSS and the mechanisms and implementation strategies of the User Access Agent System, a major component of the MDSS, are described in this paper.

Keywords: IDSS, mobile agent, MDSS.

1 Introduction

The development of Data Warehouse (DW) and On-Line Analysis Processing (OLAP) technologies has opened a new dimension for building Decision Support Systems (DSS) [10]. The Intelligent Decision Support System (IDSS) that combines the technologies of DW, OLAP, Model Base (MB), Knowledge base (KB), and data mining is a higher-level decision support system [13]. The architecture of an IDSS can be depicted in Figure 1.

In the Client layer, clients interact with the User Interface of the IDSS, which in turn interacts with the middle layer of the application servers, such as OLAP and data mining. These middle layer servers will interact with the lower layer data servers, such as the data warehouse, the knowledge base and the model base.

Although IDSS can improve the speed and flexibility of decision-making by integrating large amount of decision-making data and information in the data warehouse, it has the following deficiencies:

1. Data in the data warehouse are synthesized and stored according to the decision-making subjects and they are relatively stable. So the IDSS cannot provide a real-time integration of decision-making data for distributed decision-making.

2. IDSS cannot provide transparent accesses required by the personalized decision-making information. In an IDSS, the contents and formats of the user sharable (or accessible) decision-making information are determined by the OLAP or the data mining servers. Users have to understand the abilities and access methods of these analysis applications.
2. IDSS does not support the mobile computing mode. The emergence of the mobile computing environments has changed the ways we process and utilize distributed data. Compared to traditional distributed computing environments in fixed networks, mobile computing environments have properties such as mobility (including mobility of computing platforms, mobility of accessed resources, and personal mobility), and network adaptability (including frequent network disconnections, low reliability, asymmetrical network communications). The IDSS based on the traditional client/server model can no longer satisfy the requirements of mobile computing environments. Therefore it is necessary to introduce a new and flexible architecture.

In recent years, the agent technology has attracted attentions from many researchers. The agent technology, in particular the mobile agent technology, provides a new approach for the analysis and design of distributed and open systems in mobile computing environments. In this paper we adopt the mobile agent technology in distributed decision support systems and extend the IDSS to a mobile decision support system (MDSS).

2 Mobile Agent Technology

The concept of agents first appeared in the research of distributed artificial intelligence, in which they are defined as programs or computing entities used to model the behaviors of human beings and to provide related services, whereas mobile agents are those agents who are dynamic and distributed. That is, mobile agents are autonomous software entities with some intelligence. They can autonomously represent the user or other software entities to move among network nodes in order to search for some resources or to complete some specific functions. Because of the introduction of distributed mobile computing concepts, the mobile agent technology can successfully deal with tasks that cannot be performed well by the traditional client/server model. Agent technology has the following advantages and characteristics:

- Dynamic execution.
- Asynchronous computing.
- Parallel computation.
- Intelligent routing.
The characteristics and advantages of the mobile agent technology enable mobile agent systems to work smoothly in an environment of low bandwidth and unstable network connections, and therefore are suitable to mobile users. Introducing the mobile agent technique into the mobile computing environment, especially the mobile decision support systems, will bring a number of advantages \cite{12}, for example, it can support mobile users to carry out decision analysis in an unstable network connection situation, and can reduce the information transmission dramatically. Mobile agents can provide mobile users with personalized services, and can autonomously perform the induction according to the knowledge of network capacity and task models, relieving mobile users from knowing the capacities and access methods of individual decision support systems.

3 The MDSS Model Based on the Agent Technology

3.1 The Logical Structure of the MDSS

We can abstract the IDSS logical structures based on the traditional client/server model into a two-layered structure consisting of a Source Schema and a Client Schema \cite{8}. The source schema is also called the server schema, and in a client/server system using the "lean client machine" and the application server structure the server schema can be further categorized into two sub-schemas: an Application Schema and a Data Schema. The data schema describes the logical structure of the global data in the IDSS data warehouse; it is the global conceptual view of the IDSS. The application schema describes the logical structure of the analysis data of DSS applications (such as OLAP and DM applications) within the IDSS; it is the complete set of the decision support information schemas provided by the IDSS for users. The client schema is the sub-set (can be the complete set in special cases) of the application schema; it is actually the decision support information schemas presented to users.

Our MDSS structure adds a new layer, the agent layer, between the client and server of the IDSS. That is, it has a three-layered structure consisting of the source schema, the agent schema, and the client schema. In the MDSS, the source schema describes the global data, including data sources from the data warehouse, the OLTP database, and the external data, and the logical structure of the analysis data of DSS applications on top of the global data. The data warehouse is the main data source for the MDSS to perform its decision analysis and processing. The OLAP and data mining based on the data warehouse carry out the main tasks of information analysis. In order to be logically symmetric, we use the term of information sources to represent the data warehouse, the OLTP database, and the external data. Figure 2 compares the logical structures of the IDSS and the MDSS.
In the MDSS, the function of the agent is to access various data sources and DSS applications on behalf of the user. That is, it acts like an analyst. Therefore we represent the agent layer as the agent schema in Figure 2. The agent schema is the bridge between the application schema and the client schema; it describes a special client schema, and is the conceptual view of the application schema.

3.2 The Application Structure of the MDSS

![Figure 3. Application structure of the MDSS](image)

Figure 3 depicts the four-layered system application structure of the MDSS based on the agent technology. The application structure of the MDSS consists of the following components:

1. Data platform of the MDSS: Data platform is the DSS data sources used for the organization of analysis data. MDSS data platform includes data warehouse, the OLTP database, the external database, and other data sources, such as file system, electronic forms, E-Mail, etc. The fundamental data platform of the MDSS is data warehouse. The data warehouse integrates the original operational data decentralized in an enterprise and data received externally, and carries out most of the decision analysis.

2. Data analysis and application of the MDSS: In the application structure of the MDSS, all DSS applications are provided to the User Access Agent System (UAAS) in terms of DSS service methods. DSS service methods are relatively stable in comparison with other DSS application programs. That is, in certain time interval, the interface of a DSS service method remains unchanged. The basic data analysis of the MDSS is OLAP. Based on multi-dimensional analysis, the OLTP describes the requirements for the multi-layered, multi-angled analysis and processing of data during the management and decision-making processes. In the mean time, data mining and traditional DSS analysis are also used as the means for complementary analysis.

3. Knowledge base and model base of the MDSS: The domain base plays a crucial role in the MDSS. When an agent carries out the process of decision analysis based on multiple
information sources, it needs to use knowledge from the domain knowledge base during every step of the process. Domain base is an extended model base and method base; its contents include the global information search schema, the logical structure of the information source and stored information, the semantics of decision search and security authentication information, interfaces of the application methods for various DSS decision information applications, decision analysis rules for various information sources, etc.

4. Middleware of the MDSS: The middleware of the MDSS is the User Access Agent System. It accepts decision analysis requests from mobile users through friendly interfaces; migrates among various DSS application systems according to certain strategies; and invoking various DSS application methods to carry out decision analysis in order to obtain the information required by the user.

Compared to the IDSS in Figure 1, the MDSS has the following characteristics:

- Distribution and cooperation: MDSS is an extension of the distributed decision support system. It also requires various decision support applications to cooperate via the agent to accomplish the decision analysis.

- Mobility: The UAAS allows mobile users to access the decision support systems in the situations where the network connections are unstable and access positions are mobile.

- Autonomy: Through the UAAS, the decision support applications of the OLTP system based on data warehouse and other data sources can be independently designed, constructed, and managed.

- Adaptability and scalability: Various decision support application systems, no matter it is the OLTP based on data warehouse, the data mining, or the DSS applications based on the OLTP data sources, can be smoothly integrated with users through UAAS.
4 The User Access Agent System

The kernel of the MDSS application structure is the User Access Agent System (UAAS), depicted in Figure 4. The UAAS includes a User Agent Subsystem (UAS), an Agent Server Subsystem (ASS), and an Application Agent Subsystem (AAS).

- The UAS provides users with personalized interfaces and interacts with the user. It accepts the decision analysis requests from users and passes the requests to the ASS. The UAS consists of a User Agent (UA) and a User-access Domain Base (UDB).
- The ASS decomposes and maps the decision analysis requests from users, coordinates the decision analysis in the system on behalf of the users, and returns the results in appropriate format back to the user agent. The ASS includes the Analysis Agent (AA), Domain Agent (DA), Agent Manager (AM), Access Cache (AC) and Global Domain Base (GDB).
- The AAS is consisted of the Knowledge Agent (KA) based on the DSS data sources, the Agent Facilitator (AF), and the Local Domain Base (LDB). It is responsible for the interaction between the local DSS applications and data sources. It also provides decision analysis and processing functions for certain (e.g. local) decision supporting information.

Note that all agents in the UAAS are static agents except for the analysis agent (AA). According to the decision analysis requests from users, the AA migrates within the system and coordinates various DSS applications to perform a coordinated decision analysis to obtain the information required by users.

4.1 Description of the UAAS Components

4.1.1 Description of the Agents

An Agent in the UAAS is an active object \(^{(1)}\) of \((D, M, SD, P)\), in which \(D\) is a set of local data, \(M\) is a set of methods, \(SD\) is a set of structural dependency relationship among the methods (including trigger dependency, order dependency, real-time dependency, etc.), and \(P\) is a set of interruption and re-synchronization points.

The functions of each Agent in the UAAS can be described as follows:

- The User Agent (UA) is the interface between the UAAS and the user. It includes the following functions: (1) user login and authentication; (2) deciding the user accessible decision information resources and access interfaces, and creating personalized access interfaces according to the user requests; (3) accepting and formalizing user requests for decision analysis, and presenting these requests to the analysis agent (AA); (4) receiving the analyzing results from the AA and returning them to the user; (5) copying the knowledge required by the users from the global knowledge base when the user agent subsystem is started.

- The Analysis Agent (AA) is used to process the user requests for decision analysis. It needs to inquire from the domain agent (DA) about the processing methods for the user decision analysis requests and the corresponding domain schemas. It then uses the divide-and-conquer strategy to solve the global analysis requests from the users, and creates sub-analysis agents to execute the tasks of decision analysis. The sub-analysis agents can use remote processing or migration to interact with the knowledge agent of the service agent sub-system.
• The Domain Agent (DA) dictates the domain schemas that satisfy the user analysis requests through the access to the global domain base (GDB). It also collaborates with the knowledge agent to maintain the data consistency between the local domain base and the global domain base, and collaborates with the UA to maintain the data consistency between the user access domain base and the global domain base.

• The Knowledge Agent (KA) has the following three functions: (1) invoking the DSS application services to carry out the decision analysis on the local information sources through the original database methods and application methods; (2) learning of the local domain knowledge; and (3) maintaining the data consistency between the local domain base and the global domain base through write-through or write-back strategies.

• The Agent Facilitator (AF) is used to organize and manage the receiving, servicing, and starting of the AA when it migrates to the DSS application server, and to interacting among AAs.

• The Agent Manager (AM) is the manager of the UAAS and has the following functions: (1) the management of the security and authentication of the UAAS; (2) the service for the life cycle of agents in the UAAS, which includes the tasks for agent creation, execution, suspension, recovery, and termination; (3) the resource distribution of UAAS, including the distribution of memory and file resources; (4) position service for mobile agents, and following the positions of the frequently migrating agents in the UAAS; and (5) the management of an access cache to preserve the analysis results during network disconnections.

4.1.2 Management of the Access Cache

In order to support the access of decision support systems by mobile user machines in the case of unstable network connections, the MDSS adapts the access cache technique. That is, a data cache, managed by the AM, is attached to the agent server to temporarily store the decision analysis results of mobile users. By using this mechanism, a mobile user machine can have an unstable connection with the MDSS during the period when the UAAS is processing its decision analysis request. When the UAAS completes the user’s analysis request, it automatically stores the analysis results into the access cache. The AM will automatically return the results to the mobile user machine according to its present position when the mobile user machine logs into the MDSS again.

4.1.3 The Organization of the Domain Base for Supporting Mobile Computing

In order to support mobile computing, the MDSS organizes its domain bases in a layered architecture. A layered architecture can improve the security of the system and balance the load to the domain bases, therefore reduce the information traffic during the decision analysis and support the decision analysis search effectively in an unstable network connection situation. The organization of the MDSS domain bases is depicted in Figure 5. It has the following characteristics:

• The domain base of the MDSS is divided into three layers: the local conceptual layer, the global conceptual layer, and the user view layer. These layers correspond to the local domain bases, the global domain base, and the user-access domain bases (UDB). The domain bases in these three layers form a unified distributed domain base system. In this system, all local domain bases distributed on DSS application servers in various information sources form the local conceptual layer. They store the local information for local DSS information sources. The global domain base is stored in the agent server, forming the global conceptual layer. The user-
access domain base is the decision support schema that a current user can access and is a subset of the global domain base. The UDB also stores the information of the personalized decision access schema for the user.

- Strategies for data consistency: The data consistency in the domain bases has two aspects: the first is the knowledge consistency between the global domain base and the local domain bases, and the second is the knowledge consistency between the global domain base and the user-access domain bases.

![Figure 5. Organization of the MDSS domain bases](image)

The sources that cause the inconsistency between the GDB and the LDBs include: updating the data of the local data source causes the change of the data analysis schema; changing of the local DSS application methods; stopping the use of a local DSS application system because of certain reasons, etc. A monitoring protocol can be used to maintain the knowledge consistency between the GDB and LDBs. The concept of the monitoring protocol was originally used in the consistency maintenance for caches of multiple processors in parallel processing systems. In the design of the MDSS, knowledge agents can monitor the changes of the analysis schema in local DSS. When these changes cause a change of the system analysis schema, the KA notifies the DA to invalidate the changed analyzing subject, or to carry out an updated processing. This method is the monitoring protocol. There are two strategies for maintaining the knowledge consistency between the GDB and the LDBs: via write-through or via write-back. Because the amount of data used in maintaining the knowledge consistency between the GDB and the LDBs is relatively small, using the write-back strategy will not form a processing bottleneck when the system runs smoothly. So we adopt the write-back method in our MDSS.

The sources that cause the inconsistency between the GDB and the UDBs include: the change of the analysis schema in the GDB causes the invalidation of the existing analysis schemas of the UDB; the user uses an unregistered user machine to access the MDSS, etc. The maintenance of knowledge consistency between the GDB and the UDBs is relatively straightforward. When an analysis schema of the GDB changes, the UAAS records the contents of the change. When a user logs into the system and the UAAS discovers that there exists changed analysis schema, the changed information will be passed to the mobile user machine to update the UDB. When a user accesses the MDSS using an unregistered user machine, after passed through the authentication procedure, a new UDB is created for the user machine and the information of the user accessible analysis schema is copied to the user machine.
4.2 The Workflow of the UAAS

The workflow of the UAAS can be described in the following:

1. A user registers and invokes the local user agent subsystem (UAS). If the user is the first time to register or the access information is to be changed, the UAS copies the accessible knowledge of the user from the GDB.

2. The user agent (UA) accesses the GDB; authentication of the user and the privileges of the user; the user inputs the decision analysis request via an access interface; the UA passes the request to the analysis agent (AA).

3. The AA inquires the domain schema for the decision analysis request from the domain agent (DA)

4. The DA returns the matched information to the AA; the AA decomposes the user's global request and decides the following processing methods according to the result of the decomposition.
   - If the global request only involves one DSS application, then the AA uses the remote processing method to access the knowledge agent (KA) of the DSS application server.
   - If the global request involves multiple DSS applications, then the AA spans a number of sub-analysis agents (SAA). These SAA use the migration method to access the KAs on the related DSS application servers. The AA then becomes an agent coordinator to coordinate the communication and collaborated analysis of the SAA to complete the user's global analysis request.

5. The SAA that has migrated to the local DSS server asks the local KA to process the local analysis request; the KA first carries out the mapping of the local analysis request, producing the execution sequence of local DSS analysis methods, then the KA invokes the DSS service methods to carry out the decision analysis on the information sources in the local DSS to obtain the data or decision information required to satisfy the user decision analysis. During the process of analysis, the SAA constantly carries out collaborated operations according to the commands of the AA and communicates with other SAA. After the completion of local analysis, according to the analysis control workflow, the SAA migrates to a next DSS server or returns to the AA.

6. The AA integrates all analysis data or decision information and stores the results in the access cache of the agent server; the agent manager (AM) returns the analysis results to the proper user agent (UA) according to the position information of the mobile user found from the position service of the AM.

7. The UA returns the analysis results to the user and continues to process user requests or goes into sleep.

5 Decision-Making Based on Multiple Information Sources

The requests for decision-making submitted by mobile users in the MDSS are called global requests. The processing of these global requests usually involves multiple distributed autonomous DSS information sources. These multiple DSS information sources may have different data models, DBMSs (Database Management Systems), query languages, and even different methods for the application services. Therefore, the key problem here is to investigate how to decompose the global requests from users, transfer them into local analysis requests for individual DSS information sources and send them to these local DSSs, and then efficiently integrate the heterogeneous local
analysis results. The decision-making based on multiple information sources involves five main components, as depicted in Figure 6. They are:

- **User/program interface**: it is the main component of the user agent. It provides the users with a standard MDSS system interface. Users only need to select from the interface a set of subject analysis and analysis conditions and the interface will then transfer them into a global analysis request.

- **Global request processor**: it is the main component of the analysis agent, used to organize the whole process of the global analysis. It invokes the global request decomposer to decompose the global request into a set of local requests that are related to individual DSS information sources, and then migrates to carry out the local requests. Upon receiving all results from local analysis, it carries out the data integration and returns the results to the user agent.

- **Global request decomposer**: it is a component of the analysis agent, used to decompose a global request into a set of local requests. It needs to interact with the domain agents to obtain information on the global domain schema.

- **Request integrator**: it is a component of the analysis agent, used to integrate results from all sub-analysis according to the global domain schema.

- **Local request processor**: it is the main component of the knowledge agent for the local information source, used to map local requests into local decision analysis methods, and invokes the local analysis processor to perform the analysis.

![Decision-making based on multiple information sources](image)

**Figure 6.** Main components for decision-making based on multiple information sources

This work is different from the research in multi-database systems where a global query request is transferred into queries to individual local databases. Here we emphasize on the work of the expression, decomposition of users' global decision-making requests and mapping these requests into individual decision analysis methods for local information sources. Furthermore, because of the widely use of the Internet, users may wish to integrate not only data from databases, but also data from non-database sources, such as file systems, electronic forms, and so on. We use a connected directed graph (with a root to represent the subject) to create a generic common data...
model for supporting the integration of heterogeneous information and decision-making based on multiple information sources. In order to describe the complex referencing relationships and connection relationships among member objects, we allow cycles in the graph. We also associate descriptors to individual objects to describe those data objects that are of no display models, or those data objects that are unpredictable.

6 Remarks

The widely spread use of mobile computers requires us to provide effective means for mobile users to access enterprise information systems and the technology of decision support has become a crucial factor of information strategies of modern enterprises. However, how to construct a flexible and effective decision support system on the distributed information systems of enterprise, in order to support mobile users to access the decision support system in any time and from anywhere is still unclear. The model for constructing mobile decision support systems based on the agent technology presented in this paper provides a way to solve this problem.

We are currently building a prototype based on our model and are carrying out further researches on the model, including the learning issues of styles and interests of the user-access system, the security issues of the MDSS in a mobile computing environment, the performance issues, and so on.

In the MDSS, a user request of decision analysis (called the global analysis request) involves multiple data sources and DSS analysis applications. How to quickly and effectively decompose the global analysis requests into sub-analysis requests for various analysis sources is crucial to the performance of the whole system. It is therefore necessary to find an effective strategy to decompose a global analysis request into sub-analysis requests for individual analysis sources. We have developed a number of components and related techniques to support the decision analysis based on multiple analysis sources for the MDSS.

References


An Automatic Generation Model for Web-based Database Applications

Shui Yu   Changgui Chen   Wanlei Zhou
School of Computing and Mathematics
Deakin University
Clayton, Victoria 3168, Australia
{shuiy, changgui, wanlei}@deakin.edu.au

ABSTRACT

In this paper, we present a model by which users can customise their applications in the environment of Web-based database systems. First of all, we discuss the limitations of the current applications of Web-based database systems, then we present a component-based automatic generation model in a Web-based environment. In this model, users can travel in the database schema, choosing required attributes and database operations, such as INSERT, UPDATE and DELETE. After that, we describe two important algorithms for our model, namely the displaying of database schema and the SQL generation. At the end of the paper, we conduct a simple experiment to demonstrate the performance of the model.

KEYWORDS:
Web-based, database, graph, algorithm

1. INTRODUCTION

The growth of the Internet and the Web has increased dramatically the use of Web-based database applications. Under the Internet environment, millions of users access the Web-based database applications, and their requirements are quite different. At present, the developers of Web-based database applications develop applications according to the limited requirement analysis, hence it is obvious that the applications can not meet all the requirements of the users, who may be distributed all over the world. The developers must modify their applications even there is a small different requirement from their original requirement analysis; sometimes the developers also need to modify the programs or add new programs if there are some changes of the database objects, such as adding a new table, a new attribute to an existing table, etc. Therefore the load of the developer is heavy, and it is passive for the developer to maintain their applications to meet the dynamic requirement changes. It is necessary to develop an active method by which users can express their own requirements and customise the applications according to the changing environment.

On the other hand, most current Web-based database applications [4, 6, 7, 10] are structure-based, although they use object-oriented programming tools, such as Java, C++, etc. The structure-based application separates data and functions, most of them are organised by modules, for this reason, the components of structure-based application are difficult to be reused.

Besides those, because of the statelessness of HTTP protocol [7, 13, 14], it is difficult to flexibly control the access authorisation. Generally under the model of Browser/Server, application developers use ODBC or native drivers of the backend database product to access the backend database server, both of them need a username and password to connect to the database instance. The developers use a fixed username and password usually, otherwise, they need to develop lots of programs for each ODBC whose access authority are different. The common method threatens the security of Web-based database systems, for example, although a user does not have the authority to access a table according to the policy of administration, the user could obtain the access because he/she can access the table through the same ODBC as one who has the power to do so.

In this paper, we present a novel model, by which users can customise their database operations and the application can be generated by the model automatically according to the users' requirements. Our model also offers flexible accessing control, which remedies the shortcoming of inflexible accessing control of the current Web-based database applications. At the same time, we use the component-based method [1, 5, 8] to make our system reusable, flexible and schedulable.

The paper is organised as follows: Section 2 briefly introduces the architecture of the automatic generation model. Section 3 describes the algorithms in the model. In Section 4, we offer the performance evaluation and the comparison of source lines of code, and finally, Section 5 summarises the paper.
2. THE ARCHITECTURE

In order to overcome the limitations of the current Web-based database applications, outlined in the previous section, we present an automatic generation model, which guides users to travel among the relationships of the backend database according to the business logic. The model generates related applications dynamically, instead of developing the applications by programmer beforehand. The architecture of our model is shown in Figure 1. The system has three parts: Browser, such as IE or Netscape, running on client sites; the model, running on the same hardware of the Web server; and a database server.

![Diagram of the architecture of an automatic generation model for Web-based database application]

In the architecture, the model is independent from the database server, and it implements four functions:

1. Provides the information of database schema to users;
2. Generates SQL statements dynamically according to the requirements of users;
3. Feeds back the results to users according to their customised formats;
4. Keeps track of user status for the database accessing control.

The model is composed of several components, described as below:

- **DBExecute component.** This component creates and administers the client applets and submits SQL statements to the backend database server.
- **SELECT component.** It deals with the requirement of "SELECT" operations on the database.
- **INSERT component.** It deals with the requirement of "INSERT" operations on the database.
- **UPDATE component.** It deals with the requirement of "UPDATE" operations on the database.
- **DELETE component.** It deals with the requirement of "DELETE" operations on the database.
- **SQL Assemble component.** This component generates SQL statements dynamically.
- **DBSchema component.** This component contains two tables, relations and attributes, which are extracted from the data dictionary of the backend database, containing the tables and the relationships among the tables of the backend database.

The DBSchema component contains the database schema, and it makes the model independent from the backend RDBMS. In fact, we can obtain the information of the database schema from the data dictionary of a relational database management system directly, but it depends on the special RDBMS, therefore this method prevents the system from being used on the other RDBMS platform. In our model, the application is totally independent from the underlying relational database management system, the only job the DBA needs to do is to maintain the two tables of the DBSchema.

Now we describe the work flow of the model. When a user accesses the Web server, if there is a database-involved operation, the model will create a thread (which is running in the DBExecute component) to serve for the requirement, at the same time, a java applet is submitted from the Web server to the user's local machine. From then, all the database related requirements and results are communicated between the java applet and the thread. The thread passes the database requirement to other components to process; At last the thread in the DBExecute component receives a result (a SQL statement) from the SQL assemble component and transports it to the backend database server to execute.
The results of the database operations will be fed back to the user. More details of this work flow are listed below.

Step 1. A user starts a database request on a table of the backend database through an HTML page of the Web server.

Step 2. The DBExecute component creates a thread (DBExecute thread) for the user, and submits an applet to the user’s local machine. The applet collects the information of username and password from the user for database accessing control, and delivers the information to the DBExecute thread. The DBExecute thread uses the information to create a connection to the backend database, and keeps the connection for the user until the user disconnects it. Then DBExecute thread obtains the attributes required by the user from the DBSchema component and feeds them back to the user.

Step 3. The user chooses the attribute(s) from the feedback result. If the user needs more attributes of other tables, the applet will deliver the request to the DBExecute thread, then the DBExecute thread passes the request to the DBSchema component. DBSchema component will provide more attributes of other tables according to a schema displaying algorithm (described in section 3.1), and return the result to the DBExecute thread, then back to the user. This process will continue until no more attributes are needed.

Step 4. At this moment, the user obtains all the attributes which s/he needs and s/he is asked to decide the related database operations, such as SELECT, DELETE, UPDATE or INSERT, and the filter conditions. All of these information are collected by the DBExecute thread.

Step 5. The DBExecute thread submits the information to the SELECT component, the INSERT component, the UPDATE component, or the DELETE component according to the operation type, the respective component will organize the information to create an execute-tree based on the execute-tree algorithm.

Step 6. The SQL Assemble component obtains the execute-tree and generates the SQL statements, and returns the SQL statements to the DBExecute thread.

Step 7. DBExecute thread delivers the SQL statements to the backend database by the existing connection of the user.

Step 8. The backend database executes the SQL statements and feeds back the result to the user.

3. ALGORITHMS FOR AUTOMATIC GENERATION MODEL.

The key issues of the model are how to display the database schema to meet the potential business logic of users and how to translate the user requests to SQL statements correctly. In order to deal with these issues, we developed a set of algorithms based on the analysis of database applications. Our algorithms are based on the graph theory. We represent the E-R relationship of a database with a directed graph, and then travel on the graph to meet the logic requirements of users. In addition, the translation of user requirements to SQL statements is based on the result of the travel. In order to present our algorithms, we offer an example database to help us to explain the methods.

Suppose that there is a sports product store, which builds an E-commerce system on the Web. The tables of the database includes: State; Customer; Orders; Item; Manufacturer; Feedbackphone; Phenotype.

We express the E-R relationship of the database as a graph, where the vertices represent the entities. The edges represent the relationships between the entities. The graph for the example database is shown in Figure 2, where the names of the vertices come from the initial letter of the table names. In order to represent the graph, we define the data structure of the graph as follows.

![Figure 2. The Graph of the Example Database E-R Graph](image)

**Definition 1.** We define the data structure of a vertex as a two-element subset,

\[
\text{Vertex} = \{D, R\}
\]

Where:

- \(D = (d_1, d_2, \ldots, d_i, \ldots, d_n), \ i \in N\)
- \(R = (r_1, r_2, \ldots, r_i, \ldots, r_k), \ i \in N\)

In the definition, a vertex represents a table, \(D\) is the set of the attributes of the table, \(n\) is the number of the attributes of the relation. \(R\) represents the set of the relationships among the vertices, namely, the relationship among the tables.
3.1 The Algorithm of Schema Displaying — Partial-Breadth-First Traversal

The first step of our model is to display the schema of the backend database according to the choices of users, then the users can decide their choice on the tables and the attributes and set their filter conditions.

At the very beginning, with the help of hyperlink of HTML pages, the user can locate one table, which he/she is interested in, in other words, the user locates a vertex in the graph. Then the model will display all the attributes of the vertex. The user chooses among the attributes according to the business logic. If the user needs more attributes of other tables, the java applet will submit this message to the model. The model starts from the current vertex and travels all adjacent vertices. As a result, a set of vertices are obtained. After that, the model displays all the attributes of the vertex subset to the user for selection. Sometimes, this process continues until the user needs no more attributes.

It is obvious that breadth-first traversal [2, 11, 12] of the graph is suitable for the model. However the user may not choose all the vertices in the vertex subset, that means sometimes it is not necessary to travel all the vertices which are adjacent to the current vertex. Based on this practical situation, here we present a modified algorithm of the breadth-first travel algorithm, which is named as partial-breadth-first traversal.

The work follow of the partial-breadth-first travel is listed bellow,

Step 1. We construct two queues for the traversal, and two pointers are used for the queues. One is the current pointer, and another one is the alternative pointer, and the queues are named as current queue and alternative queue, respectively.

Step 2. We put the first vertex, which is chosen by the user through the HTML page, in the current queue.

Step 3. Use the breadth-first traversal to travel the adjacent vertices of the current vertex or vertices, which is (are) in the current queue, and put the result of the travel into the alternative queue.

Step 4. Display all the attributes of the vertices in the alternative queue.

Step 5. Delete the vertices, which have no attributes selected by the user, from the alternative queue.

Step 6. Exchange the pointers of the two queues.

Step 7. Set the alternative queue empty, and go to step 3 until the end of the traversal or the user stops it.

After the traversal we obtain a subgraph, which represents the requirements of the user for a database operation. Most of the time the subgraph is a tree, which we name as execute-tree.

3.2 The Algorithm of SQL Generation

Once we obtain the execute-tree, we can create the SQL statement based on the execute-tree and the other information, such as the operation type,

![Figure 3. An Example of Execute-tree](image)

the filter conditions, etc. Figure 3 shows an example of an execute-tree.

In the execute-tree, there are three fields in every node. The V_i represents the vertex in the graph, namely, V_i represents a table; the second field is D_i, a set of the chosen attribute(s) of the vertex by the user; the third field is C_i, which represents the filter condition(s).

We can generate SQL statements, such as SELECT, INSERT, UPDATE, or DELETE, based on the execute-tree. Among the four types of database operations, the SELECT operation is used most frequently, therefore we only present our algorithm for the SELECT operation here.

Step 1. Displays all the attributes of the execute-tree to the user, collects the conditions and the logical relationships (AND, OR, NOT) among the conditions, which are added by the user to the C_i field(s).

Step 2. Travels the execute-tree using the Preorder Traversal Algorithm, and finds all the items in the D_i field of each node to complete the SELECT clause of the SELECT statement, uses the contents of the V_i fields to complete the FROM clause and assembles the WHERE clause using the information of the C_i fields.

Step 3. Travels the execute-tree using the Preorder Traversal Algorithm again to obtain the join conditions based on the edges of the execute-tree, and uses them to complete the WHERE clause.

4. PERFORMANCE ANALYSIS

Performance is an important issue for Web-based database applications. All SQL statements and the interfaces of the model are generated automatically and dynamically. For this reason, we conducted a simple experiment to try to find the difference on performance
between applications based on the model and the traditional Web-based database applications, whose SQL statements and interfaces are written manually. We conducted the experiment in a LAN. The configuration is listed below:

Database server: Personal Oracle 8i running on MS Windows 98,
CPU: 200M
Memory: 64M
Web Server: MS Personal Web Server running on the same PC of database server
Browser: IE5.0

The SELECT operation is the most complicated and time-consuming operation. In the model, the SQL statements, which are generated automatically and dynamically, usually include a number of JOIN operations. Therefore we test the performance of SELECT operations with JOIN operation(s). The result is shown in figure 4.

![Figure 4. Performance comparison](image)

In Figure 4, we find that the difference on performance between the two kinds of applications is very small. One important reason is that most of the current mainstream commercial RDBMS products provide optimizers [3] to optimize the SQL sentences before their execution.

5. SUMMARY

Generally speaking, it is impossible that the original design for a Web-based application will satisfy all types of requirements of the users, who may be dispersed all over the world, and the dynamic changes of the environment. Hence, in this paper, we develop a model for Web-based application users to define their requirements and result formats. The model is component-based, and it is independent from the backend database product. In particular, we transfer the E-R relationships of a database to an undirected graph, and propose some algorithms based on the graph to display the database schema and assemble SQL statements. The performance of our model is almost the same as manually developed Web-based database applications. This benefit comes from the relational database management system, which can optimize the SQL sentences before their execution. The model makes the developer free from modifying or adding codes to meet the new requirements from time to time, and it also improves the security of Web-based database applications.

REFERENCES:

FAILURE MONITORING IN A REPLICATED AND MOBILE ENVIRONMENT

JIANTING ZHONG
jzt@deakin.edu.au
School of Computing and Mathematics
Deakin University
Geelong, Victoria 3217, Australia

WANLEI ZHOU
wanlei@deakin.edu.au
School of Computing and Mathematics
Deakin University
Geelong, Victoria 3217, Australia

Abstract Mobile transaction management in a replicated environment is a difficult task due to the long lived transaction complexity, which results from the large scale of network, and mobility requirement, which invokes consistency handling among replicas located in both static and mobile hosts. When site failure and network partition problems are taken into consideration, it forms the most complicated system environment in mobile computing. This paper reports a solution of using Failure Monitor as a separate system component in each subset of the whole network system to manage transaction processing in the presence of site failure and network partition. The transaction management model is presented first. Detailed algorithms focusing on the functions of Failure Monitor are described according to different failure scenarios.

1. INTRODUCTION

Replication technology in a distributed system is the key to providing high availability, fault tolerance and to enhancing performance[9]. Most of the replication control protocol reviewed in[2-6,8] are designed without considering site failures and network partitions.

A mobile transaction management system usually covers a large scale of network consisting of different LANs. Transactions involving mobile hosts like laptop computers are called "mobile transactions". Such a system is characterized by the mobility control and its normally long lived transaction type requiring collaboration between sub-transactions within one single transaction request. The mobility control includes keeping track of the mobile host, which can issue transactions from different subnet, and guaranteeing consistency of the replicated data object in both static and mobile hosts. Site failures and network partition though may be rare, they are realistic problems which can not be neglected. System design, including all the three elements, replication control, mobility control and failure handling, is challenging and easy to go into pure theoretical side which is hard to be implemented.

We propose a relatively centralized method to solve the problems of site failures and network partitions in our mobile transaction management system using the Primary/Non-primary replication control method. The idea is stimulated by the protocol introduced in [1] yet our Failure Monitor has many more functions than their Voltaire Witnesses. The Failure Monitor in our method is placed in every subnet and is responsible to deal with failures as a dedicated server. The method aims at simplifying the communication type between different servers during the presence of failures.

The rest of the paper is organized as follows. Section 2 presents our system environment (for more details see ref. [11]). Section 3 describes our algorithms. Section 4 concludes the paper.

2. TRANSACTION MANAGEMENT MODEL

We use fully replicated server with identical functions to manage data objects distributed among different sites interconnected by a communication network. Some sites can disconnect from base stations time to time. We classify them as mobile hosts. They have their independent processing power and the access to data objects stored locally. A transaction request issued by a client, either from a static host or a mobile host, can consist of different sub-transactions each of which is to be serviced by a group of servers called replicas. A transaction manager who receives the request from client divides the transaction into sub-transactions and passes onto the different replicas. Among one group of replicas, a Primary Replica leads other Non-primary Replicas. We use server group members to refer to both transaction manager and replica components for transaction execution.

Without generosity we assume all the transactions are update oriented. The transaction processing policy is to treat the Primary Replica for every sub-transaction, or service, as the check point for fully commit mode. Any replica can execute a service freely but a partial commit mode is returned if it is a Non-primary Replica. Only those transactions checked by Primary Replica will be finalised by either upgraded to a fully commit mode or downgraded to an abort if conflict exists. This covers the mobile host in which only Non-primary Replicas are located. Coordination between replica groups are carried out by Transaction Manager to finalise transaction after collecting result from different service execution.

Our network environment is tailored to different subnets connected by gateways, same as described in [1]. Network partition happens when gateway between subnets fails. This leads to a situation where server group members distributed in different subnets can not communicate and may stop a transaction processing. We add a dedicated Failure Monitor as a server component in each subnet to find out the partition existence and to help in transaction processing.

For simplicity, we include two subnets connected by one gateway in our network configuration. The following
FIGURE 1: SYSTEM ARCHITECTURE

In each subnet, we have server groups which are comprised of Primary Replicas and Non-primary replicas. For transactions involving different services, different groups of replicas with different PR leading several NFRs are responsible. The client issues transactions through transaction manager (TM) which sits on top of the server groups. The TM for PR site is TMPR, while for NFR site is TMNFR. A mobile host, denoted as MH below, also has a transaction manager TMmbm and the servers are Non-primary ones, NFRmb.

TM is the centre for transaction processing, as described before, while FM is the centre for failure handling. It either contacts the TMs in the same subnet, or contacts other FMs in other subnets.

3. DETAILED ALGORITHM DESIGN

We request that each system component be aware of the addresses of the other parties which it needs to communicate. Specifically to an MH, once connected to the base station, it registers itself to an administration engine and gets the necessary addresses of other parties. Since our system is transaction based, transactions exist in different states, either partially committed or fully committed, flowing to and from different component. This requests that all the server group members and the FMs maintain different object lists for recording different transaction types. We describe our algorithms under different failure scenarios in the next sections.

3.1 NP SITE FAILURE

The site which includes Non-primary servers is denoted as NP site. It can be down in two scenarios.

* Scenario One

An NP site is down when the client initializes transaction and sends it to the site. In this case, the client will find out by a time out mechanism and can simply choose another site to execute the transaction[2]. If the server on an MH is down, the user should wait for the server being repaired if it is in the disconnection mode; or it can reconnect to the base station to execute transactions using other static servers. (We use "base station" interchangeably with "static host" in this paper.)

* Scenario Two

An NP site failure can be detected when the site is contacted by a P site to force the execution of a transaction. When the P is aware of the specific NP site failure, it records the address of that NP together with a unique transaction identifier and the TMPR sends a message to the FM in its subnet to inform it the situation. We use a NP_site_failure_found() function to show the communication from the TMPR to the FM informing the NP site failure.

```
NP_site_failure_found()
| set failure site address;
| set failure type as NP site failure;
| set transaction identifier; /* which the failed site has handled */
| store the transaction in a waiting_for_execution queue for that site;
| send the message object to the FM;
| LIST 2: NP_site_failure_found()
```

Before the TMPR decides to contact the FM, it should first check to see if the unreachable site is an MH. This requests the TM be aware of the MHs as special group members. If the TMPR finds that the unreachable site is an MH, it does not contact the FM knowing that the MH is in a disconnection mode. Instead, it simply stores the transaction in the queue for the MH and sends it to the MH for execution later on when the MH reconnects to the base station.

3.2 P SITE FAILURE

The site which includes a Primary server is denoted as a P site. This is detected by an NP site who sends transaction initialized by a client and the transaction has been partially committed expecting the P site to finalize it. Different NPs could detect the P site failure at the same time. They all should report to the FM who decides the new P site and informs all the members within that server group. After receiving the new P site notice, each server member should first check to see if itself is set as the new P site. If it is, it invokes the functions for being the P site which previously was masked.

The following functions illustrate the above ideas.

```
P_site_failure_found() /* for NP who detects the P site failure report to the FM */
| set message object();
| set P site address as the failure site address;
| set failure type as P site failure;
| store the partially committed transaction for further finalisation;
| send the message object to the FM;
| LIST 2: P_site_failure_found()
```

```
new_primary_set()/* invoked by the FM when it finds out the P site failure is being reported */
| select a NP site to be the new P site;
| change the P site setting in that transaction server group list;
| broadcast the new P site to its server group members;
| LIST 3: new_primary_set()
```

```
new_setting_check() /* for each NP site when receiving the new P site setting from FM */
| (if the new P site is itself)
| invite the functions to be a P site;
| store the P site address; /* this is kept as part of the network view need to be update constantly*/
| LIST 4: new_setting_check()
```
When an NP site is upgraded to a P site, it commits all the transactions in the wait_for_finalising transaction queue which previously partially committed and sends them to all the other NP site for execution. Other NP sites should send their wait_for_finalising transactions to the new P site after being notified by the FM of the P site change.

For an MH which is disconnected from the network when the P site failure happens, it is notified of the network view which includes all the updated P site addresses for different transactions stored by the FM when the MH reconnects to the base station. This guarantees the MH to send out its partially committed transactions executed locally during its disconnection to the right place.

3.3 NETWORK PARTITION

We discuss this problem from three phases: Detecting, Notifying, and Resolving.

- Phase One: Network Partition Detecting

This can be done by two procedures running simultaneously. The first procedure lets an FM at different subnet regularly broadcasts message to other FMs to see if they are reachable. If an FM does not reply within a maximum time frame, the gateway between the two FMs is assumed down, which leads to the two subnets being partitioned from each other. The second procedure lets an FM receive certain amount of reports from server group members that some other members are not reachable. The FM decides that the gateway might be down by noticing those unreachable members are all located in the same subnet. For this purpose, separate counter for each relative subnet is maintained by the FM for failure report. Then the FM specifically sends messages to the FM in that subnet to confirm if the partition situation is true or not. If the target FM does not reply in time, the network partition is considered to be true. The FM reset the failure report counter for that subnet and goes into the phase two for notifying. We use partition detecting() functions to illustrate the idea.

partition_detecting_regular() /* regular detecting between different FMs */
{ while (true) /* always executing */
  get time frame for regular gateway checking;
  set all the FMs as the broadcasting targets;
  send out checking message wait for reply;
  if any FM does not reply then record it;
  invoke the partition_notifying procedure;}
LIST 6: partition_detecting_regular()

partition_detecting_by_failure() /* noticing by site failure amount in one subnet */
{ while (true) /* always executing */
  if receives site failure report then increase a failure counter for that subnet;
  check the failure counter for each subnet;
  if a failure counter exceeds certain upper limit then find the FM address in that subnet;
  set that FM as the message sending target;
  send out checking message and wait for reply;
  if the FM replies on time then set the failure counter for that subnet as zero;
  if the FM does not reply within certain time frame then invoke the partition_notifying procedure;}
LIST 6: partition_detecting_by_failure()

- Phase Two: Network Partition Notifying

Once the network partition has been detected, the FM is responsible to notify all the server groups about the situation to save unnecessary network communication overhead by server members trying to contact the other partitioned subnet. The FM is also responsible to notify all parties once the crashed gateway is up and the partition no longer exists.

- Phase Three: Network Partition Resolving

During the network partition, the main situation to handle is that a client could issue a transaction request which involves server members in different partitioned subnets. We first assume all the P sites for one such transaction locate in the same subnet, which is the normal case in most transactions. This is set as Scenario One below. We then expand such situation to more complicated ones to be discussed later in Scenario Two.

Scenario One: In this case, network partition could happen in two cases. One is when P site sends a transaction to NP site in the other subnet. The other is when NP site sends a transaction to P site in the other subnet for checking and finalizing it from a partial commit mode. In either case, these transactions are all sent to FM for recording and further processing. When the partition situation no longer exists, different FMs exchange their knowledge of transactions for further proceeding. The reason we choose the FM to store such transactions is to prevent transactions from lost if the site which initializes the transactions fails during a network partition. When FM receives a transaction record during a partition, it identifies its type, whether initialized by other P/NP sites, then stores it in different object list. We use to_other_FM() function for FMs to exchange their recorded transactions after the gateway between them is back to normal.

to_other_FM()
{ send the transaction object list to the other FM;
  block waiting for incoming messages;
  if an incoming message is the confirmation of receiving transactions from the other FM then delete the transaction object list and break;
  if an incoming message contains the transaction list initialized by other P/NP sites then store the transactions and send confirmation to the sending FM;"
LIST 7: to_other_FM()

We use partition_transaction_execution() function to illustrate how the PNP sites further proceed with transactions received from FM.

partition_transaction_execution()
{ receive transaction list from FM;
  mask itself as busy; /* transactions from FM have priority to be checked first */
  check the transactions by order until the list is empty
  if the transaction is from P for compulsory execution
    execute the transaction;
    check to see if it conflicts with present state
    if conflict exist then invoke certain conflict resolving program;
  if the transaction is from NP for checking and finalizing
    check to see if it can be executed;
    if conflict with the present state
      abort the transaction; notify the original NP site to roll back; /* this is the primary first policy to guarantee the primary site interest*/
      contact the original NP to finalize the transaction; /* no conflict */}

539

591
The conflict resolving program can either abort the transaction or expose the conflict to the user expecting manual resolve according to different application system [7].

Scenario Two: We now consider the situation where network partition results in different P sites involved in one transaction cannot communicate. In this case, the TMRR which receives the client request simply can not fully execute the whole transaction. We propose two options for the TMRR to offer to the client. One option is to let the client abort the transaction and the other is to store the transaction and re-execute it after some time frame specified by the client hoping the network partition no longer exists then.

3.4 MHT HANDLING WHEN NETWORK PARTITIONED

We use MHT to denote a transaction initialized in an MH and has been partially committed during the period that the MH is disconnected from the base station. We first assume that the MHT does not involve sub-transactions served by other servers located in the base station. When the MH is reconnected to the base station, the MHT needs to be sent to some P site for checking and finalizing. At this point, if the subnet is partitioned by gateway failure and the targeted P site for the MHT is in another subnet, some methods should be carried out to handle it.

In our proposal, the MH should be notified about the network partition once it connects to the base station. The MH, after being notified, submits its MHT to the FM in stead of the unreachable P site for storing. The MHT is set as a special transaction type compared to those initialized from static hosts. When FM sends the MHT to the P site after the partition disappears, P site will identify the MHT and the final result (either fully committed or abort) is kept until the MH receives it. The difference between an MHT and a normal transaction lies in the fact that the MH may disconnect from the partitioned subnet before the gateway is up. It results in the time for the MH to receive the finalising notice from the P site to be hard to decide depending on the next reconnecting time of the MH. We propose that the MH should actively contact the P site to get notice of the MHT finalising every time it reconnects to the base station.

We finally consider the scenario that an MH reconnects to the base station and issues transactions involving sub-transactions served by some unreachable static server group members due to network partition. It is proposed that the MH should choose to abort the whole transaction requesting FM to purge the MHT before it moves on.

4. CONCLUSION AND FUTURE WORK

We have presented our design of using a dedicated server, the Failure Monitor, in each possibly partitioned subnet connected by a gateway to deal with site failures and network partitions in a mobile transaction management system with the Primary/Non-primary replica model. Compared to the Volatile Witness system in [1], our Failure Monitor is not required to record any transaction during normal cases. We also introduce how the FM handles the mobility issues in mobile transaction management when failure occurs. This has not been discussed in [1] where it does not include a mobile computing environment.

However, one may argue that a relative centralized Failure Monitor itself is fault prone. This situation could be very rare by placing Failure Monitors in stable sites. While the possibility does exist, transactions recorded by Failure Monitors should be backed up in non-lost devices when they are sent by other PNP sites. Regeneration technique for Failure Monitor in this case could be discussed. Further work of system modeling and simulating is being carried out to evaluate the performance of our failure monitoring design.

5. REFERENCES

A Web-based Design For The Mobile Transaction Management of A Distributed Database System

Jianing Zhong and Wanlei Zhou
School of Computing and Mathematics
Deakin University
Pigdons Road, Geelong
Victoria 3217
Australia
{jnz,wanlei}@deakin.edu.au

Abstract

A mobile transaction management system enables users to receive transaction based services from both static and mobile hosts distributed in a large scale of network. The difficulty lies in the heterogeneity of computer systems and in the communication control to guarantee concurrency and consistency of transaction operations. The World Wide Web offers a uniformed vehicle to interface with users ubiquitously and an ideal architecture to implement mobile transaction systems. This paper reports a design using Java, JDBC and Web browser to implement a distributed database system, particularly a transaction management system covers mobile hosts and static workstations/PCs in which data and servers are replicated.

Keywords: Transaction Management, World Wide Web, Distributed Database System

1. Introduction

A mobile transaction management system usually covers a large scale of network consisting of different LANs. Transactions involving mobile hosts like laptop computers are called "mobile transactions". Such a system is characterised by the mobility control and its normally long lived transaction type requiring collaboration between sub-transactions within one single transaction request. The mobility control includes keeping track of the mobile hosts, which can issue transactions from different subnets, and guaranteeing consistency of the replicated data object in both static and mobile hosts. Site failures and network partition though may be rare, they are realistic problems which cannot be neglected. System design, including all the three elements, replication control, mobility control and failure handling, is challenging and easy to go into pure theoretical side which is hard to be implemented. Apart from that, users in such a system may use different computer hardware and software systems. An efficient communication control method is needed to accommodate heterogeneity in both hardware and software.

The popularity of World Wide Web (WWW) makes it possible for users everywhere to access and possibly process information distributed in the network. Web well supports the Client/Server computing model which is most appropriate for interacting with a transaction management system. Java as a fully object-oriented, network based language is thought to be the best technology today for dynamically loading applications onto client systems [1].
Putting together technologies such as Java applets, JDBC and Web browsers enables users to do transactions on distributed databases and get feedback through the Web in an interactive and consistent way.

We have used the Primary/Non-primary Replica (P/NP) technology to build a mobile transaction management system which can access fully replicated databases. The databases, including Oracle, Mysql and Access, are stored in both static workstations/PCs and laptop computers in Ethernet LANs connected by gateways. We identify that the tasks of managing mobile transactions using the primary and non-primary replicas can be classified into three levels [16]:

- **Level One.**
  - single service, the service has many replicas,
  - a reliable primary replica, and
  - a mobile unit with one non-primary replica.

- **Level Two.**
  - multiple services, each service has many replicas,
  - reliable primary replicas and reliable links among primary replicas, and
  - a mobile unit with many non-primary replicas.

- **Level Three.**
  - Same as level 2, but primary replicas can fail and links among primary replicas can fail (site failure/network partition).

We propose solutions to all the three levels and present a Web-based design for implementation. The rest of the paper is organised as follows. Section 2 presents our transaction processing model (more details refer to [16,17]). Section 3 describes our design for Web-based implementation. Section 4 summarises the paper and outlines our future work.

2. **Transaction Processing Model**

2.0 Replication Method Description

We are specifically interested in the merging of mobility and replication method and adopting such method in a distributed database system for managing transactions. Existing replication control algorithms [3-16] basically fall into the following categories:

- whether the strict consistency among replicas are released or not ("being lazy" or not)
- providing fault tolerance (site failures/network partition) or not

Our proposed algorithms release the strict consistency among replicas and provide solutions to site failures/network partitions which are more likely to happen in a mobile environment due to the wider range of network heterogeneity.

2.1 Solutions to Level One/Two Transaction Processing

We use fully replicated server with identical functions to manage data objects distributed among different sites interconnected by a communication network. Some sites can disconnect from base network time to time. We classify these sites as mobile hosts. They have replicated data objects stored locally and have independent processing power. A transaction request issued by a client, either from a static unit or a mobile host, can consist of different sub-transactions, each of which is to be serviced by a group of servers called replicas. A
transaction manager who receives the request from the client divides the transaction into sub-transactions and passes onto different replicas. Among one group of replicas, a Primary Replica leads other Non-primary Replicas. Primary Replica can only sit on the static units and Non-primary Replica can sit on either static or mobile units. We use server group members to refer to both the transaction manager and replica components for the transaction execution. Figure 1 illustrates the relationship between each server group member in transaction processing.

Without generosity we assume all the transactions are update oriented. The transaction processing policy is to treat the Primary Replica for every sub-transaction, or service, as the check point for a fully commit mode. Any replica can execute a service freely but a partial commit mode is returned if it is a Non-primary Replica. Only those transactions checked by Primary Replica will be finalised by either upgraded to a fully commit mode or downgraded to an abort if conflict exists. This covers the mobile host in which only Non-primary Replicas are located. Coordination between replica groups is carried out by Transaction Manager to finalise transactions after collecting results of different service execution. If there is only one service involved in a transaction, only one replica group is needed. This is the case of Level One transaction processing model defined in Section 1. The arrows to "other Replica group" in Figure 1 is not needed then. If there are multiple services involved in a transaction, different replica group are needed as shown in Figure 2. This is the case of our Level Two transaction processing model.

2.2 Solutions to Level Three Transaction Processing

2.2.1 General Description: As stated in the introduction, our network environment is tailored to different subnets connected by gateways. Network partition happens when an intersubnet gateway fails. This leads to a situation where server group members distributed in different subnets can not communicate and may terminate a transaction processing. We add a dedicated Failure Monitor as a server component in each subnet to check out the partition existence and to help in transaction processing.

For the sake of simplicity, we include two subnets connected by one gateway in our network configuration. Real system which include larger scale of subnet distribution can be easily extended based on our model.

Gateway is used to link different subnets. We have one Failure Monitor (FM) for each subnet. FM is a dedicated fault tolerance program sitting on stable unit. Static hosts, workstations or networked PCs, are distributed in different subnets. The solid lines are used between the static hosts and the subnets indicating the connection between them is static.
Mobile Host can move around. We use dashed lines between the Mobile Host and the subnets indicating the mobile connection mode. TMs/PRs/NPRs are all server processes sitting on different hosts according to certain application environment. Yet as stated before, only Non-primary Transaction Manager and Replica can sit on the Mobile Host. Client can issue transaction request through Transaction Manager (TM) from anywhere in the whole network. For one single transaction with only one service, one leading Primary Replica (PR) and several Non-primary Replicas (NPRs), which could be in different subnets, are responsible for it. For transactions involving different services (multi-services), different groups of replicas with different PR leading several NPRs are responsible.

**Figure 2: System architecture**

Figure 2 shows our system setting. Among all the system components, it is the Transaction Manager at each site that contacts the Failure Monitor when failures happen. Different FMs contact each other for detecting network partitions and exchanging transaction records when a partition is resolved.

The TM is the center for transaction processing, as described before, while the FM is the center for failure handling. The communication between different components can be illustrated by Figure 3. The FM either contacts the TMs in the same subnet, or the other FM in the other subnets. The communication types involving the FM are:

- accept failure report from the TM which detects failures during a transaction execution
- inform the TMs within a server group about a primary site failure in that group
- inform all the TMs in the subnet about the network partition
- talk to the FMs to detect network partition
- exchange transaction records with another FM when the network partition between them recovers
2.2.2 Identify Failures: Site failures can happen in both Primary sites and Non-primary sites. A network partition happens when a related gateway is down so the involved subnets can not communicate with each other.

Our key idea is to use a Failure Monitor in each subnet as a centralized point to deal with both site failures and network partitions. Its responsibility under different failure scenarios is identified as follows.

- **Non-primary Site Failure (NP Site Failure)**
  Get information from the component which detects the failure and signals the system administrator to fix the problem.

- **Primary Site Failure (P Site Failure)**
  Get information from the component which detects the failure. Inform the members in that group which the failed site is leading about the failure. Choose another site to be the primary site and inform all.

- **Network Partition**
  Detect the partition either actively, by regularly talking to other FMs in other subnets, or passively, by noticing that some server site failures all happen in the same subnet. After detecting the network partition, the FM informs all the server group members in its subnet to save further useless network traffic to the unreachable subnets during the partition. At the same time, the FM records all the transactions from the server group members. When the gateway is up and the network partition no longer exists, the FMs at different subnets exchange their transaction records and send these records to the target server group members for further proceeding.

3. Implementation Design for A Distributed Database System

3.1 The Web-based Design

Our transaction management design does not make any assumption of underline systems. It potentially can be implemented in any platform which supports multi-threading (for concurrency) and message passing. However, the popularity of the Web and the availability of HTTP present a readily available platform for our model in which system components and clients interact with each other in a large scale of network [2]. Java embedded Web browsers allow users to easily fetch Java programs at run time. We choose one such browser as the interface between user, which is the front end, and our transaction management system which covers both static and mobile units and all the system components (TM/Primary Replica/Non-primary Replica/FM) are Java implemented programs. Java applets in a Web page, triggered by a user when viewed by the Web browser, are links to execute certain
transaction operations. In the back end, we use Java Database Connectivity (JDBC) for the servers in the transaction system to access the physical data sources in various locations.

Java and JDBC, from our system prototyping experience for building a trading system over heterogeneous databases, have been proven to be an ideal tool for network computing. Their flexibility of coding once and running everywhere nicely suits our needs for heterogeneous environment [3]. Specifically, when we are working on a distributed database system, JDBC frees developers by its goal of being a DBMS independent interface, a uniform interface to different data source. It actually completes part of the work of schema translation which is the key issue dealing with different database systems. When contacting the actual data source, all one has to do is to address the URL (Unified Resource Location) in the network. The lower level data retrieving and manipulating is handled by JDBC drivers. From the user's point of view, that means one single query sentence can be used to access data in all kinds of sources, Oracle, MySql or Access, etc., provided the system being used has installed specific JDBC drivers.

Figure 4 shows the information flow in our Web-based design for the transaction management in a distributed database system [4].

![Figure 4: Information flow in our design](image)

Users who have access to local Web servers use the (Java enabled) Web Browser to locate a Web page designed for our system. There are Java applets embedded in the page containing fields for user input and for displaying feedback. After selecting his/her transaction service, the user sends the request through the Web which actually triggers the execution of server programs on the fly. Transaction Servers in the above figure generally refer to those transaction management components which are Transaction Manager, Primary Replica and Non-primary Replica (TM/PR/NPR). JDBC is the bridge between servers, which are all Java programs, and different databases. JDBC services are called by the server programs embedded with JDBC driver objects.

On the user's side, once the user knows the Web site he/she is able to go into the transaction service system no matter which kind of machine he/she is using. If the user is carrying a mobile host, he/she can connect to the local Web server to get service done. In the Web page, we have three kinds of fields. One field is the list of transaction services for the user to choose. The user also can choose different servers for the sake of certain performance requirements. The second field is for the user to input service request according to the information obtained from the first list. The user is asked to provide his/her identification before the request can proceed. The third field is the area for displaying the outcome of the transaction execution. The applets are remote Java programs on the client side. When the user goes into the Web page, the applets are downloaded into the local machine and run for the user.

On the Transaction Servers' side, we use Java's Remote Method Invocation (RMI), a powerful tool for structuring distributed client/server application [1]. The TMs in our system are remote server objects running for incoming calls from applets all the time. They have the detailed information of the server groups which actually execute individual subtransactions included in a user's request. When a TM receives a call from the applet, it first decides which server groups to contact. It then acts as a secondary client to invoke those
server groups using RMI. After the result is returned to the TM, it passes them on to the applet to display to the user and do the necessary job of coordinating among replicas. The server groups are the Primary/Non-Primary Replicas which are stand alone Java programs sitting on different machines. The JDBC driver objects are embedded in these programs.

The distributed databases included in our system are shown in Figure 5 [16]. We have three different Database Management System (DBMS), Oracle, mSQL, Access, running on different workstations/PCs connected by a 10M Ethernet LAN. It shows that the heterogeneity in a mobile environment includes different DBMS, operation system and also hardware. All the system components, TM/PR/NPR, are distributed on different static/mobile units and can be easily configured to change their site, e.g. move from one station to another.

Our Web-based design using all Java implementation overcomes the system heterogeneity well. It forms a strong backbone of data sources for users distributed in different networks to issue transaction requests and get feedback at run time.

![Figure 5: Architecture of databases in a mobile environment](image)

3.2 Design for Disconnection Mode of Mobile Host and the FM

The Web-based design introduced in the above section using the Web Browser as the user interface is the normal case in our system. However, there are other situations in which user does not have access to the Web if he/she is using a mobile host in a disconnection mode. Apart from that, the system fault tolerance issues using FM are transparent to the user level. Accordingly, we make separate design relative to mobile host and the FM in our system.

3.2.1 Mobile Host Disconnection: When the mobile host is disconnected from the network so that it has no access to the Web, the user uses a Java application which has the same function of the applet program. The TM and the server group in this case are all local programs. After the mobile host reconnects to the network, the local TM does the job of contacting the other TMs in the network to keep the system consistent. The point here is the Web page designed is for most users which normally has the access to the Internet/Intranet and the mobile users which have access to the networks. For the mobile host’s disconnection mode, we use Java application which later on still has the ability to communicate with remote server objects once the host is reconnected. This is because in our server level, all components are distributed Java programs.

3.2.2 The FM Design: The FM is designed only to handle the failure situation in the server
level which the communications between different server components are hidden from the end user. The network administrator interfaces with the FM and the users need not to deal with it. So we design the FM as a stand-alone Java program. They are not relative to the apples introduced in the above section.

4. Summary and Future Work

The transaction management model proposed in this paper deals with various problems including replication control, mobility control and fault tolerance. We adopt the model into a distributed database system. The implementation design using Web-based tools, specifically Java relative technology, is proposed aiming to handle the heterogeneity problem and the network communication issues which characterise our distributed and mobile system.

We have identified three levels in our transaction model. At the present stage, we have built a prototype database system which includes Web page embedded with Java applets and server programs embedded with JDBC. It proves the feasibility of our whole Web-based design. The system implements the first and second transaction level which could handle both single service and multi-service transactions. Preliminary result of experiments have shown that the system meets our fault tolerance target in a replicated and mobile environment without putting too much communication overhead. The efficient prototyping and promising outcome have convinced us the charming of Java technology and encouraged us to further implement our system towards the Level Three model.

References


