I am the author of the thesis entitled

A Reactive System Model for Building Fault-Tolerant Distributed Applications

submitted for the degree of

Doctor of Philosophy

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

Full Name................................Changgui Chen..................................................

(Please Print)

Signed ...........  [Signature Redacted by Library] ...........................................

Date..............................  /2/16/2001..................................................
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></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
A Reactive System Model for Building Fault-Tolerant Distributed Applications

by

Changgui Chen
M.E. (University of Science & Technology of China, China)
B.E. (Shanghai Jiao Tong University, China)

Submitted in fulfillment of the requirements for the degree of
Doctor of Philosophy
School of Computing and Mathematics
Deakin University

Supervised by: A/Prof. Dr. Wanlei Zhou

July 2001
I certify that the thesis entitled:

A Reactive System Model for Building Fault-Tolerant Distributed Applications

submitted for the degree of:

Doctor of Philosophy

is the result of my own research, except where otherwise acknowledged, and that this thesis in whole or in part has not been submitted for an award, including a higher degree, to any other university or institution.

Full Name.......................... Changgui Chen...........................................

(Please Print)

Signed........... Signature Redacted by Library..............................

Date......................... 12/11/2021..................................
Acknowledgment

First of all, I would like to sincerely thank my supervisor, Associate Professor, Dr. Wanlei Zhou for initiating this research project, providing lots of suggestions and various helps throughout the whole period of my PhD study. He taught me lots of things about the research on distributed and fault-tolerant computing area, which I feel so prospective now, and encouraged me to think, speak and write in English instead of Chinese, so that I could concentrate on my research quickly without the language problem. After I started writing my thesis, he reads the draft thoroughly many times and corrects my numerous manuscripts. I am fortunate to have been able to work with him since the beginning of my PhD study.

I am grateful to School of Computing and Mathematics, Deakin University, which has provided an excellent environment for learning and experiencing. Also, my gratitude goes to the Head of School, Professor Andrzej Gosciniski who has been giving great support in granting generous allowances to postgraduates for attending conferences. Those conferences I have attended both domestically and overseas have broadened my knowledge and mind enormously.

I thank our school secretaries, Rai Sahib and Kathy Giulieri, for their warm smiles and efficient assistance. I would also like to thank all my friends and colleagues in Deakin for their various activities and helps to make me having no time to think about the loneliness during my staying in Deakin.
Abstract

The development of fault-tolerant computing systems is a very difficult task. Two reasons contributed to this difficulty can be described as follows. The first is that, in normal practice, fault-tolerant computing policies and mechanisms are deeply embedded into most application programs, so that these application programs cannot cope with changes in environments, policies and mechanisms. These factors may change frequently in a distributed environment, especially in a heterogeneous environment. Therefore, in order to develop better fault-tolerant systems that can cope with constant changes in environments and user requirements, it is essential to separate the fault tolerant computing policies and mechanisms in application programs. The second is, on the other hand, a number of techniques have been proposed for the construction of reliable and fault-tolerant computing systems. Many computer systems are being developed to tolerant various hardware and software failures. However, most of these systems are to be used in specific application areas, since it is extremely difficult to develop systems that can be used in general-purpose fault-tolerant computing.

The motivation of this thesis is based on these two aspects. The focus of the thesis is on developing a model based on the reactive system concepts for building better fault-tolerant computing applications. The reactive system concepts are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms. The stress of the model is to provide flexible system architecture for the general-purpose fault-tolerant application development, and the model can be applied in many specific applications. With this reactive system model, we can separate fault-tolerant computing polices and mechanisms in the applications, so that the development and maintenance of fault-tolerant computing systems can be made easier.
Publications

Papers published by international conferences during my Ph.D study:


Contents

Abstract........................................................................................................i

List of Figures..........................................................................................ix

List of Tables..........................................................................................xi

List of Listings..........................................................................................xii

Chapter 1  Introduction.............................................................................1

1.1 Distributed Systems..............................................................................2

1.2 Fault-tolerant Computing .................................................................3

1.3 Current Trends for Fault-tolerant Computing ...................................5

1.4 Reactive System Approach...............................................................6

1.5 Aims of the Thesis................................................................................7

1.5.1 Building the Reactive System Model .........................................8

1.5.2 Implementing and Evaluating the Reactive System Model ...........9

1.6 Thesis Layout ...................................................................................10

Chapter 2  Overview of Reactive Systems .............................................13

2.1 What is a Reactive System ................................................................13

2.2 Concurrency Issues ..........................................................................16

2.2.1 Definition ...................................................................................17

2.2.2 Concurrency and Reactivity .....................................................17
Chapter 2 Methods in the Development of Reactive Systems ...15

2.2.3 Processes and Threads .......................................................18
2.2.4 Synchronization ..................................................................19
2.3 Specification Methods .................................................................20
  2.3.1 State Transition Systems ....................................................21
  2.3.2 Statecharts ......................................................................22
  2.3.3 Temporal Logic ..................................................................24
2.4 Reactive Programming .................................................................26
  2.4.1 The Asynchronous Approach ...............................................26
  2.4.2 The Synchronous Approach ................................................27
  2.4.3 Asynchronous-Synchronous Coupling: Esterel + CSP. ..........28
  2.4.4 Distribution Method ..............................................................28
2.5 Related Work -- Tools for the Development of Reactive Systems ......28
  2.5.1 STATEMATE .................................................................29
  2.5.2 An Object-Oriented Method -- DisCo .................................29
  2.5.3 PSP Model .....................................................................30
  2.5.4 Structured Reactive System Notation .................................32
  2.5.5 Agent-based Reactive Systems ...........................................33
2.6 Summary ..............................................................................35

Chapter 3 Modeling and Design .........................................................37

3.1 The Generic Reactive System Architecture ..................................38
3.2 Some Concepts .....................................................................40
3.3 Actor and Agent Model ............................................................43
  3.3.1 Distributed Object Methodology .......................................43
  3.3.2 Actor Model ....................................................................44
3.3.3 Agent Model ................................................................. 45
3.3.4 Messages and Protocols .............................................. 46
3.4 Components Modeling .................................................. 48
  3.4.1 DMM Agent ............................................................. 48
  3.4.2 Sensor Actor ........................................................... 51
  3.4.3 Actuator Actor ......................................................... 53
3.5 Reactive Behaviors ..................................................... 55
  3.5.1 States and Transitions ............................................. 55
  3.5.2 Behaviors of DMM Agents ....................................... 57
  3.5.3 Behaviors of Sensors ............................................... 59
  3.5.4 Behaviors of Actuators ........................................... 60
3.6 System Modeling ......................................................... 61
  3.6.1 Safety and Liveness ............................................... 63
  3.6.2 System Architecture ............................................... 66
3.7 Summary ................................................................. 67

Chapter 4 Communication Services ..................................... 69
4.1 Processes in a Reactive System ..................................... 70
4.2 Correctness Requirements ........................................... 72
4.3 Ordering Constraints ................................................... 74
  4.3.1 Ordering Definitions ............................................. 75
  4.3.2 Assigning Ordering Constraints ............................... 77
4.4 Ordering Protocols ...................................................... 78
  4.4.1 Implementing the FIFO Ordering ............................... 78
  4.4.2 Implementing the Causal Ordering Constraint ............... 79
Chapter 4.4.3 Implementing the Total Ordering Constraint ................................................. 80
4.5 Reliable Multicasting Service ............................................................................. 82
  4.5.1 Atomic Multicast ......................................................................................... 82
  4.5.2 Multicasting Atomicity Protocol ................................................................. 83
4.6 Membership Management ............................................................................... 86
4.7 Summary ........................................................................................................ 88

Chapter 5 Implementation Issues ....................................................................... 90
  5.1 System Framework ......................................................................................... 91
  5.2 Communication Patterns ............................................................................ 94
    5.2.1 Multicast Datagram Communication .................................................... 94
    5.2.2 Stream-based Communication ................................................................ 95
  5.3 Implementing Temporal Constraints ............................................................ 96
  5.4 Simple and Composite Entities .................................................................... 98
  5.5 Java DMM Agent .......................................................................................... 99
    5.5.1 Multicast Datagram DMMs .................................................................... 100
    5.5.2 Stream-based DMMs .............................................................................. 104
  5.6 Java Sensor Actor ......................................................................................... 106
    5.6.1 Multicast Datagram Sensors .................................................................. 107
    5.6.2 Stream-based Sensors ........................................................................... 109
  5.7 Java Actuator Actor ...................................................................................... 111
    5.7.1 Multicast Datagram Actuators ............................................................... 111
    5.7.2 Stream-based Actuators ......................................................................... 112
  5.8 Properties Checking ....................................................................................... 114
  5.9 Summary ....................................................................................................... 114

vi
Chapter 8  Conclusion and Future Work.........................157

8.1  Summary of the Thesis.................................................157

8.2  Main Contributions.................................................160

8.2.1  Modeling and Design of Reactive Systems..................160

8.2.2  Generic Java Reactive System Package ...................161

8.2.3  Performance Analysis............................................161

8.3  Future Work..........................................................162

References........................................................................164
List of Figures

Figure 2.1: A reactive system with its interface set 14
Figure 2.2: A device model for reactive system 31
Figure 2.3: The structure of reactive control systems 32
Figure 2.4: The structure of an agent-based reactive system 34
Figure 3.1: The generic reactive system architecture 39
Figure 3.2: The structure of a DMM agent 49
Figure 3.3: A sensor actor 51
Figure 3.4: An actuator actor 53
Figure 3.5: The behavior of a DMM agent 57
Figure 3.6: The behavior of a sensor actor 58
Figure 3.7: The behavior of an actuator actor 60
Figure 3.8: The definition of a reactive system 61
Figure 3.9: The reactive system structure 66
Figure 4.1: Communication in a reactive system 71
Figure 4.2: Message delivery in a reactive system 74
Figure 5.1: The Layered Framework 90
Figure 5.2: The Object Layers 91
Figure 5.3: Multicast datagram communication 93
Figure 5.4: Tunnelling multicast packets between subnets 94
Figure 5.5: The generic sender architecture
Figure 5.6: Sensors and actuators
Figure 6.1: The ARTs with the multicasting communication
Figure 6.2: The ARTs on three groups of tests
Figure 6.3: The multicasting and stream-based testing on a local host
Figure 6.4: Multicasting and stream-based testing
Figure 6.5: The effects of the ordering constraints
Figure 6.6: Event sensor and Timing sensor tests (1)
Figure 6.7: Event sensor and Timing sensor tests (2)
Figure 6.8: The execution times for different compositions
Figure 6.9: Two groups of simulations
Figure 7.1: A distributed replication system
Figure 7.2: Using polling sensors
Figure 7.3: Using event sensors
Figure 7.4: Using the DMMs with active replication policies
Figure 7.5: Structure of a replication system
Figure 7.6: The partition-tolerant system architecture
Figure 7.7: The partial ordered sub-tasks
Figure 7.8: Information flow for supporting teamwork
Figure 7.9: The architecture of the teamwork coordination
List of Tables

Table 6.1: The ART (ms) tested on a local host. 122
Table 6.2: The ARTs (ms) of three types of sensors (single subscriber) 128
Table 6.3: The execution times (s) for the simulation 132

List of Listings

Listing 4.1 -- Vector Timestamp Protocol 78
Listing 4.2 -- Assigning the USN Protocol 80
Listing 4.3 -- Atomic Multicasting Protocol 84
Listing 5.1 -- Temporal Constraint Protocol 96
Chapter 1

Introduction

A distributed computing environment provides a lot of advantages than a centralized system. High speed networks make it possible of running distributed software on multiple machines efficiently [Stallings 1998]. With the ever-growing dependency being placed on the distributed computing system, the requirement for reliability of the system is increased enormously. A number of techniques have been proposed for the construction of reliable and fault-tolerant systems. Many computer systems are being developed to tolerate various hardware and software failures. However, most of these systems are to be used in specific application areas, since it is extremely difficult to develop systems that can be used in general-purpose fault-tolerant computing.

The focus of this thesis is on developing a model for building better fault-tolerant computing systems. The model is based on the reactive system concepts, which are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms. The stress of the model is to provide a flexible system architecture, which separates fault-tolerant computing polices from fault-tolerant computing mechanisms, for the general-purpose fault-tolerant application development.

In the following sections, we introduce the context of this thesis. In particular, we give the definitions of basic concepts involved, such as distributed systems,
Chapter 1. Introduction

fault-tolerant computing, and reactive systems, etc. Then we address the aims of this thesis and outline the thesis structure.

1.1 Distributed Systems

A distributed system is a system consisting of a collection of autonomous machines connected by communication networks and equipped with software systems designed to produce an integrated and consistent computing environment [Coulouris 1994]. Distributed systems enable people to cooperate and coordinate their activities more effectively and efficiently. The key purposes of the distributed systems can be represented by: resource sharing, openness, concurrency, scalability, fault-tolerance and transparency [Goscinski 1991].

- **Resource sharing.** In a distributed system, the resources – hardware, software and data can be easily shared among users. For example, a printer can be shared among a group of users.

- **Openness.** The openness of distributed systems is achieved by specifying the key software interface of the system and making them available to software developers so that the system can be extended in many ways.

- **Concurrency.** The processing concurrency can be achieved by sending requests to multiple machines connected by networks at the same time.

- **Scalability.** A distributed system running on a collection of small number of machines can be easily extended to a large number of machines to increase the processing power.

- **Fault-tolerance.** Machines connected by networks can be seen as redundant resources. A software system can be installed on multiple machines so that in the face of hardware faults or software failures, the faults or failures can be detected and tolerated by other machines.

- **Transparency.** Distributed systems can provide many forms of transparency such as: location transparency, which allows local and remote information to
Chapter 1. Introduction

be accessed in a unified way; failure transparency, which enables the masking of failures automatically; and replication transparency, which allows duplicating software/data on multiple machines invisible.

Applications of distributed systems range from the provision of general-purpose computing facilities for groups of users to automated banking and multimedia communication systems, and they embrace almost all commercial and technical applications of computers [Daniel 1993].

1.2 Fault-tolerant Computing

Computer systems may fail due to hardware and software faults, such as processor, storage device, power outage, software design errors, etc. In many cases, such failures may have disastrous results, for instance, in an airline traffic control system, a system fault may result in plane crashes [Cristian 1996]. With the ever-growing dependency being placed on distributed systems, the requirement for reliability of these systems is increased.

When failures occur in hardware/software, the system may generate incorrect results or may simply stop before finishing the intended computation. Therefore, failures in distributed systems can have different semantics, and in turn, they require individual treatments [Cristian 1991] [Jalote 1994]. Distributed systems are typically subject to two kinds of failures: site failure and communication link failure, which can result in the following failure semantics:

- **Fail-stop failure** [Schlichting 1983]. Fail-stop failure is used to describe a process/processor, which either works correctly, or simply stops working without taking any incorrect action. The fail-stop process/processor has the property of informing others by a notification service upon the failure or remaining in a state that the failure is detectable to others. There is also another term Fail-silent failure.

- **Network link failure** [Tanenbaum 1996]. This refers to the breakdown of a
communication link between sites. The link failure makes it impossible to send or receive messages over the failed links. Also messages in transmission can be lost.

- **Network partition failure** [Birman 1996]. Network link failures can lead to partition failure, where a group of sites involved in a distributed system is partitioned into a set of subgroups, of which members of the same subgroup can communicate with each other but not with members of different subgroups.

- **Timing failure** [Johnson 1989]. This refers to a violation of assumed temporal property of the system, such as clock drift bound between machines, or a message transmission delay between sites linked by networks.

- **Byzantine failure** [Lamport 1982]. This type of failures refers to any violation of the system behavior. In particular, it is used to refer to corrupted messages, such as malicious messages, that give wrong instruction, and as a result, may bring down the system.

*Fault-tolerant computing* is defined as the correct execution of a specific algorithm in the presence of defects [Toy 1987] [Zhou 2000]. At the highest level, fault-tolerant computing systems are categorized as highly available and highly reliable. Most fault-tolerant applications can be described either as being transaction based or as being concerned with process control [Coulouris 1994]. The factor that distinguishes the two is the recovery time. Transactional services can generally accept occasional failures followed by a relatively lengthy recovery procedure.

Process control applications have different requirements, and they can be modeled with *reactive system* concepts [Jalote 1994]. They are characterized by having inputs that are readings taken from sensors and outputs to actuators that are either used to control a process directly or to activate alarms so that humans can intervene in the process. Applications of this type include air traffic control, monitoring patients in hospitals and controlling reactors. They generally have
very strict timing requirements, and, therefore, recovery must be achieved within a very small time limit.

1.3 Current Trends for Fault-tolerant Computing

The development of fault-tolerant computing systems is a very difficult task. Many researchers have conducted extensive work on the fault-tolerant computing area since 1980's. These work are extremely useful, and the current trends of them can be summarized as follows:

- **Algorithms** [Zhou 1995(a)]. One area of research topic is to search for the proper fault-tolerant algorithms and models for various types of applications.

- **Architectures** [Siewiorek 1986] [Stallings 1993]. Many researches are now carried out in searching for the best fault-tolerant architecture for general-purpose computing as well as for specific applications.

- **Systems** [Cristian 1991]. Many computer systems are being developed to tolerate various hardware and software failures. Most of these systems are to be used in specific application areas, since it is extremely difficult to develop systems that can be used in general-purpose fault-tolerant computing.

- **Reliability evaluation** [Siewiorek 1992]. As computer systems become more and more complex, reliability modeling and evaluation is now an extremely hard task.

Among these trends, developing computer systems is a straight way to provide fault-tolerant computing services. Many specific computer systems, such as The Air Traffic Control System [Cristian 1996], The Tandem Computer System, etc., have been developed with the fault-tolerant services. However, fewer efforts are made for developing the general-purpose fault-tolerant computing systems, since it is difficult and lacks of developing tools.
Chapter 1. Introduction

As mentioned above, process control computer systems can be modeled with the reactive system concepts. In fact, many fault-tolerant applications can be regarded as reactive systems. Such a system has a flexible architecture and can be used in the general-purpose fault-tolerant computing.

1.4 Reactive System Approach

The concept of reactive systems [Harel 1985] was introduced by D. Harel and A. Pnueli to describe systems that are supposed to maintain an ongoing interaction with their environments, rather than to produce some final result on termination. They propose a dichotomy between what they call transformational and reactive systems. On the one hand, transformational systems are the traditional ones, which accept inputs, perform transformations on them and produce outputs. Generally speaking, transformational systems are well described by a relation between input and output values. They perform input/output operations, perhaps prompting a user from time to time to provide extra information. On the other hand, reactive systems are more complex. A reactive system is repeatedly prompted by the outside world, and its main role is to react continuously to external inputs by producing outputs, rather to obtain a final result. More generally, a reactive system does not compute or perform a function, but is supposed to maintain a certain ongoing relationship with its environment. Typical examples of reactive systems include flight reservation systems, industrial plant controllers, operating systems, most kinds of real-time computer embedded systems, Web servers, and communication systems, etc.

The reactive system concepts are an attractive paradigm for system design, development and maintenance because it separates policies from mechanisms [Zhou 1997]. 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. The DMMs and sensors/actuators in
Chapter 1. Introduction

the reactive system can be separated without impact on each other. This feature will have a great significance in developing distributed and fault-tolerant applications. The major advantage of the reactive system model in fault-tolerant computing is the separation of the mechanisms for gathering information and dealing with failures from the policies for making decisions according to the information received, and therefore can lead to a better software architecture.

The reactive system concepts also have stronger semantics in the real world [Chen 2000(a)]. The traditional programming method is to decompose a system and its structure into smaller or manageable objects which are usually passive. It may cause some problems in developing modern systems because they may consist of active objects. The reactive system approach views the whole system as a reactor and it consists of independent active objects or actors. Actors can initiate activities and respond to stimuli from the environment, hence they may reflect real objects in the world. Compared with previous programming approaches, the reactive system method is more close to reality and has stronger semantics.

1.5 Aims of the Thesis

Developing fault-tolerant applications is an extremely difficult task indeed. One of the reasons is that, in normal practice, fault-tolerant computing policies and mechanisms are deeply embedded into most application programs, so that these application programs can not cope with changes in environments, policies and mechanisms. These factors may change frequently in a distributed environment, especially in a heterogeneous environment. Therefore, in order to develop better fault-tolerant systems that can cope with constant changes in environments and user requirements, it is essential to separate the fault tolerant computing policies and mechanisms in application programs. This can be achieved by using the reactive system approach, which has a great advantage that is the separation of polices and mechanisms.
Chapter 1. Introduction

By using the reactive system model, we can separate fault detection and fault tolerance processing from applications. We want to show 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. The feasibility and initial advantages of using reactive system concepts in building Java distributed applications have been shown through our recent work [Chen 2000(a)] [Chen 2000(b)] [Chen 2000(c)].

Several systems, such as Meta [Wood 1994], Disco [Systa 1991] and STATEMATE [Harel 1990], and languages, such as Reactive C [Boussinot 1991] and Reactive Pascal [Quintero 1996] that are based on the reactive system concepts have been developed recently. However, most of the researches on reactive systems are concentrated on process control (such as controlling a robot). In our research, we plan to apply the reactive system concepts in developing distributed applications, particularly fault-tolerant applications. Hence, the main aim of the thesis is to develop a model based on the reactive system concepts for building better fault-tolerant distributed applications. Specifically, we will build and implement the reactive system model for developing general-purpose fault-tolerant applications. According to this aim, the thesis is divided into two stages: (1) modelling and design of reactive systems, and (2) implementation and evaluation of the design.

1.5.1 Building the Reactive System Model

The thesis will make a significant contribution to the distributed and fault-tolerant computing theories and applications in general. The first aim of the thesis is to build the formal reactive system model, which is the theoretical basis for the research. To model reactive systems, we introduce the Actor concept, which was originally proposed by Hewitt [Hewitt 1977], and later developed by Agha [Agha 1986]. Actors are parallel autonomous agents, which are distributed in space and execute at their own rate and communicate asynchronously by sending messages. Actor is the basic concept in the Real-time Object-Oriented Modeling which
Chapter 1. Introduction

provides a framework for modeling reactive systems, because most real-time systems are reactive systems in nature [Selic 1992] [Selic 1994].

Using the reactive system concepts in fault-tolerant computing is new. We have to translate the concepts from process control or real-time applications into the fault-tolerant computing environment. In particular, we will build models for concepts such as events, failures, time constraints, distributed objects, sensors, actuators and decision making managers. To detect and tolerate failures, we have to understand the nature of the events to be monitored, the nature of the failures to be reported, and the ways of communications among sensors, actuators, decision making managers and application objects. These understanding will form the basis of the entire system design.

1.5.2 Implementing and Evaluating the Reactive System Model

The second aim of this thesis is to implement our design. The reactive system model we build in the thesis will be implemented as a software package using the Java programming language. 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 1997]. As most reactive systems are distributed systems, we can use the Java language to implement the reactive system model as a generic Java package, in distributed environments. This generic Java package consists of a set of distributed object classes that form the practical basis for developing fault-tolerant Java applications.

As a reactive system consists of decision making managers, sensors, actuators and application objects, the thesis will achieve the following goals in particular:

- A generic decision making manager class that implements some fault-tolerant policies. In particular, the following replication based fault-tolerant policies [Guerraoui 1997] and their variations are to be implemented by various
Chapter 1. Introduction

decision making managers: primary-backup replication policies, and active replication policies (the definitions of these concepts will be given in Chapter 5, same next).

- A generic sensor class that can monitor and report the states of distributed objects to its subscribers. Using such a generic sensor, many types of sensors, such as event sensors, timing sensors, and polling sensors, can be implemented.

- A generic actuator class that can change the states of applications. Using such a generic actuator, many types of actuators, such as warm start-up actuators and cold start-up actuators can be implemented.

The evaluation on the efficiency and performance will be given after the implementation. The evaluation includes a series of experiments, which are conducted on different groups to compare their results for the evaluation of the system efficiency and performance, and several applications of the reactive system model in fault-tolerant computing environments, for the purpose of demonstrating the potential benefits of our Java platform.

1.6 Thesis Layout

There are eight chapters in this thesis. The structure of the rest of the thesis is organized as follows:

Chapter 2: Overview of Reactive Systems, comprehensively investigates the reactive systems and their inherent characteristics, as well as the related work on the development of such systems. This chapter introduces what a reactive system is, what characteristics it has, specification methods, reactive programming, and current tools for the development of reactive systems. These concepts will form the basis for the proposal of the generic reactive system model.

Chapter 3: Modeling and Design (of Reactive Systems), proposes a generic reactive system architecture model based on the concepts reviewed in Chapter 2,
and introduces the Actor and Agent concepts to model reactive system components: decision making manager (DMM), sensor and actuator; and builds the reactive system model formally. As the reactive system architecture model is a component-based model, we try to use Actor and Agent to model sensor/actuator and DMM respectively. We consider a reactive system as a distributed computing system consisting of a number of Actors and Agents. Their behaviors comprise of the overall behavior of a reactive system.

Chapter 4: *Communication Services*, discusses the group communication services used in the reactive system model; particularly atomic multicast, ordered communication and group membership management are addressed in this chapter. To ensure the reactive system model design in Chapter 3 correct and fault-tolerant, we invoke the atomic multicast protocol and three ordering constraints: FIFO, causal ordering and total ordering. The atomicity protocol guarantees the complete communication in the system, i.e., either all members receive a message from a sender or none of them receives. The ordering constraints ensure the message passing is ordered in the system.

Chapter 5: *Implementation Issues*, addresses the issues of implementing the reactive system. The implementation of the reactive system model involves many issues such as communication patterns, various reactive control protocols, etc. The DMM class, sensor class, and actuator class are implemented using the Java programming language. Multicast datagrams and stream-based communication are achieved respectively. Various constraint protocols, such as ordered constraints and temporal constraints, are implemented in the encapsulations of these classes.

Chapter 6: *Performance Study*. After implementation we have conducted a series of tests to evaluate the performance of the Java reactive system package. The tests are performed in different groups with different network environments to compare their results for the evaluation of the system efficiency and performance. Also, a simulation of the reactive system model is presented to evaluate the performance in the case that it consists of a great number of sensors,
Chapter 1. Introduction

DMMs and actuators.

Chapter 7: Applications. Three application examples are studied in this chapter – 1. The Web-based Teamwork support system; 2. The network partitioning problem; and 3. The Web-based information system. The goals of these systems are to guarantee that the systems provide continuous services even in the case of a server failure, a site failure, or a network partitioning failure. These applications will demonstrate the potential benefits of the Java reactive system.

Chapter 8: Conclusion and Future Work, summarizes our work, addresses the major contributions of this thesis, and points out the future work.
Chapter 2

Overview of Reactive Systems

Chapter 1 introduced that the aim of this thesis is to develop the reactive system model for building better fault-tolerant computing applications. Our first task, therefore, is to investigate the reactive system concepts, such as what a reactive system is, what characteristic it has, and what the current trend is for such systems, etc. There have been a great deal of literatures published on the development of reactive systems, but many of them are focused on process control. Our research is totally different from those works and is a fresh topic in distributed computing. We will elaborate our work in the subsequent chapters. In this chapter we do some reviews on the reactive system concepts and investigate the current trends for the development of such systems. These concepts will form the basis for our proposal of the generic reactive system model.

2.1 What is a Reactive System

A reactive system has been defined by D. Harel and A. Pnueli as one that is supposed to maintain an ongoing interaction with its environment, rather than to produce some final result on termination [Harel 1985]. It cannot be described adequately only as computing a function from an initial state to a terminal state. An adequate description of reactive systems must refer to their ongoing behaviors,
Chapter 2. Overview of Reactive Systems

which are seen as reactions to external stimuli. A reactive system continuously interacts with its environment, using inputs and outputs that are either continuous in time or discrete. The inputs and outputs are often asynchronous, undeterministic, meaning that they may arrive or change values unpredictably at any point of time. Such systems are often concurrent and distributed [Boasson 1993] [Broy 1997].

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, whether normal user-generated or environment-generated or abnormal ones (e.g. failures), by sending signals and commands to its environment [Glaesser 1996].

Figure 2.1 depicts a reactive system paradigm, where M represents the system control; e1, e2, e3 ... (depicted as arrows) represent a set of events between the system and its environment (inputs and outputs), which comprise the system interface E (={e1, e2, e3, ...}); S represents the system behavior which description characterises the desired behavior of the system using the elements of E. The interface (E) of the system may contain descriptions of those input and output channels, signals, requests and responses, that constitute the system’s interaction with its environment.

---

Figure 2.1: A reactive system with its interface set
Chapter 2. Overview of Reactive Systems

From Figure 2.1, we can see that a reactive system is repeatedly prompted by
the outside world, and its main role is to react continuously to external inputs (like
e1, e2, ...) by producing outputs (like e3, e4, ...), rather to obtain a final result.

Generally speaking, a reactive system does not compute or perform a
function, but is supposed to maintain a certain ongoing relationship with its
environment, or in other words, a reactive system mainly focuses on the
interactions between the system and its environment. It receives information from
the environment, and then makes decisions to produce outputs to react to the
environment. The primary purpose of such a system probably is to support its
users in the difficult process of decision making, often in some difficult situations
and under severe time constraint [Huizing 1992]. The environment information,
by which the system will make the decision to control the system behavior, will
be acquainted through sensors. The decision made by the system will be sent to
the environment through actuators [Boasson 1996].

Reactive systems usually display unpredictable, often catastrophic, behaviors
under unanticipated circumstances. Owing to their inherently complexity, reactive
systems have many uncertain features. For example, they can be deterministic or
not, terminating or not, required to respond in real-time or not. Also, they can
contain concurrently or sequentially executing components, and the cooperation
of their components can be required to be synchronous or asynchronous, etc.
Typically, reactive systems illustrate the following main characteristics [Caspi
1994]:

- **Concurrency (Parallelism).** Reactive systems are generally composed of
  concurrent communicating and executing components or sub-processes, either
  for geographical requirements, or for safety and fault tolerance criteria. These
  systems are often implemented on parallel architectures. It is convenient and
  natural to design such systems as sets of parallel components that cooperate to
  achieve the intended behaviors.

- **Determinism.** Determinism is an important characteristic of reactive systems.
  We say that a system is deterministic if it produces identical output sequences
Chapter 2. Overview of Reactive Systems

when fed with identical (timed) input sequences; otherwise, it is non-deterministic. Most real-time reactive systems are non-deterministic, therefore they are very hard to deal with. In this thesis we only discuss deterministic reactive systems. Deterministic systems always react the same way to the same inputs. Purely sequential systems are obviously deterministic. However, determinism does not mean sequentiality. Determinism makes the system design, analysis and debugging easier than non-deterministic one.

- **Temporal and logical requirements.** These requirements concern both the input rate and the input/output relationship and response time. They are induced by the environment and must imperatively be matched. Logical properties involve the input/output specification, and temporal properties involve the timing constraints. Both properties must be expressed in the specification, be taken into account during the design, and their satisfaction must be checked on the implementation (to prove their logical and temporal correctness).

The design and construction of reactive systems cannot be carried out without a clear understanding of the systems’ intended behaviors [Kurki-Suonio 1993]. The reactive system behaviors are very complex. Their description requires specifying the relationships of inputs and outputs over time. Such descriptions may involve complex sequences of those external input and output events, actions, conditions, and information flow, often with explicit timing constraints, that combine to form the system overall behaviors.

### 2.2 Concurrency Issues

Most reactive systems are generally composed of concurrent communicating and executing components or sub-processes, either for geographical requirements, or for safety and fault tolerance criteria. Concurrency gives rise to problems that are quite different from the ones sequential systems present, and they form the
most difficult tasks faced in the development of reactive systems. We focus our discussion on the concurrency issues in this section.

2.2.1 Definition

Concurrent systems are ones in which various activities are in the process of being executed at the same time, that is, concurrently [Bacon 1993] [Burns 1993]. When these activities exist in a single computer we can say that they are executed concurrently. If the computer is equipped with only a single central processor, this is achieved by interleaving the execution of portions of each activity or process. If the computer has multiple processors, then various processes or activities can be executed simultaneously, that is, in parallel or parallel execution.

In a distributed environment, there are many computers, each with one or more processors. If there are $M$ computers in a distributed system with one processor each, then up to $M$ processes can run in parallel, provided that the processes are located in different computers. As reactive systems are often distributed and concurrent systems, we may call them concurrent or parallel reactive systems.

A number of processes or activities may be in progress simultaneously within a concurrent system, either because components of a concurrent algorithm have been started off in parallel or because a number of users may simultaneously make demands on the system. The concurrent system may be an operating system, a sub-system within an operating system or an application or service running above an operating system. In all cases, the concurrent system may be distributed across a number of computers rather than centralized in one computer.

2.2.2 Concurrency and Reactivity

The nature of a reactive system is in its reactivity. Reactivity characterizes the nature of interaction between the system and its environment. It states that this interaction is not restricted to accepting inputs on initiation and producing outputs
on termination. In particular, it allows some of the inputs to depend on intermediate outputs. Concurrency, on the other hand, refers to the internal organization of the system. One important link between concurrency and reactivity is that a component in a concurrent system should always be viewed as a reactive component, independently of whether the entire system is reactive or not. This is because, typically, a component in a concurrent system maintains a reactive interaction with other components in the system [Pnueli 1986]. Consequently, we regard all concurrent systems as reactive if we view a full system in the way we view its components.

Concurrency and parallelism give rise to non-determinism because a system can have several distinct behaviors and it must choose one of them arbitrarily [Manna 1981]. Usually, parallel reactive systems consist of a number of concurrent and parallel processes, and display a non-deterministic behavior. These processes continuously interact with each other and cooperate to achieve a pre-defined goal. A typical parallel reactive system contains at least two controllers that control many physical devices. The behavior of the entire system depends on the reaction between controllers and the controlled devices.

2.2.3 Processes and Threads

A concurrent system contains various processes which are executed in parallel. A process is a running program, which consists of an environment for execution together with at least one thread of control. A thread is the operating system abstraction of an activity. The definition of a thread can be described as follows. The overall behavior of a system can be broken down into a sequence of more primitive behaviors or steps, each of which can be comprehended individually. Each step transforms the state of the system by executing a specific action. The overall result of the system behavior can be deduced by following the sequence of steps from beginning to end. Such a sequence is called a thread of control, since it controls the behavior of a computer. A portion of a system that
Chapter 2. Overview of Reactive Systems

incorporates a sequential thread of control is referred to as a process or a task. A process may have one or more threads of control.

A reactive system contains two or more simultaneous threads of control (e.g. a sensor thread and a control thread) that dynamically depend on each other, or interact with each other, in order to fulfill their individual objectives. This interaction between threads is at the heart of most difficulties in dealing with concurrent systems. Specifying the behavior of concurrent systems requires simultaneous awareness of multiple entities and their progression relative to each other, as well as with respect to absolute time.

The two primitive forms of interaction between threads are:

- **Synchronization.** Synchronization involves adjusting the timing of the execution of an action step in a thread based on the execution state of other threads. It may be required either to achieve noninterference between threads (mutual exclusion) or to ensure proper interaction between them.

- **Communication.** It is often necessary to pass information from one thread to another. This can take on many different forms, including global shared memory, message passing, remote procedure calls, and rendezvous.

The combination of these two forms of interactions forms synchronous communication, in which synchronization and information exchange occur simultaneously, and asynchronous communication, in which information exchange does not directly imply synchronization.

### 2.2.4 Synchronization

A concurrent system contains a number of processes. These processes may have several simultaneous inputs or outputs and they may need to pass data to one another or to communicate. Since the speed at which each process works is not determinable, non-determinism arises in linking the processes. For instance, even if several processes begin together they may not all end simultaneously. Therefore, we say that there is a race condition and the parts of computation
Chapter 2. Overview of Reactive Systems

containing such processes are time critical. A need therefore arises to introduce a mechanism to coordinate and control the temporal order in which processes are executed to realize a given parallel algorithm. Such a mechanism is called a synchronization mechanism.

Two distinctly different methods are used for synchronization among processes:

- **Shared-variable method**: in this method, synchronization and communication among the processes are achieved using shared mechanisms under a centralized control.

- **Message-passing method**: in this method, the processes are autonomously controlled in sending and receiving messages, without sharing data; they are coordinated by using delay and wait operations among them.

These two methods have contributed to the development of new styles of programming languages for concurrent programming.

2.3 Specification Methods

The development of reactive systems starts from the specification of the systems and their behaviors. There have been a number of methods proposed for the specification of reactive systems [Barroca 1998] [Gerth 1997] [Glaesser 1996] [Harel 1987] [J cUvinen 1990] [Schenke 1994]. Furbach has investigated four types of specification methods for reactive systems [Furbach 1993], namely the state transition systems, Petri nets, logic-based systems, and programming language-oriented systems. However, the widely accepted and well-known methods for specifying the reactive system are Statecharts, which is based on state transition systems, and Temporal Logic.
2.3.1 State Transition Systems

The description of reactive system behaviors involves the sequences of states, actions, conditions, and information flow, often with explicit timing constraints. Hence, a reactive system can be described by the state transition system (also called state machine or state automaton) model [Pinna 1995] [Pnueli 1986]. A labelled state transition system is represented by a quadruple

\[ M = (S, A, \rightarrow, T) \]

where

- \( S \) – A set of states (possibly infinite).
- \( A \) – A set of actions.
- \( \rightarrow \) – A transition relation (a relation on states for each action).
- \( T \) – A set of initial states (\( T \subseteq S \) and \( T \neq \emptyset \)).

- \( A = I \cup O \), \( I \) is a set of input actions; \( O \) is a set of output actions.

- Let \( a(s) \) be an action \( a \) on the state \( s \). If \( a(s) \neq 0 \) we say that the action \( a \) is enabled on the state \( s \). If \( a(s) = 0 \), then \( a \) is said to be disabled on \( s \). Let \( G \subseteq A \) be a set of actions. If for some \( a \in G \), \( a \) is enabled on \( s \), we say that the set \( G \) is enabled on \( s \).

- A transition relation is a family of relations: \( \rightarrow \subseteq S \times A \times S \), or \( \xrightarrow{a} \subseteq S \times S \), for each action \( a \in A \). It is common and convenient to write transitions in the form

\[ \xrightarrow{a} \sigma', \sigma \in S, a \in A. \]

This proposition expresses that in the state \( \sigma \) the considered system may perform (is ready to do or accept) the action \( a \) and then (after having done the action \( a \)) be in the state \( \sigma' \).

Given a transition system \( M \), we define a computation of \( M \) to be a finite or infinite sequence of states and transitions:
Chapter 2. Overview of Reactive Systems

\[ \omega: s_0 \xrightarrow{a_0} s_1 \xrightarrow{a_1} s_2 \xrightarrow{a_2} \ldots \]

satisfying the following requirements:

- Initiality: \( s_0 \in T \)
- Consecution: For each \( i, s_{i+1} \in a_i(s_i) \)
- Termination: \( \omega \) is finite and terminates in \( s_k \) only if \( s_k \) is terminal, i.e., for every \( a \in A, a(s_k) = 0 \).

We define that the behavior of a state transition system \( M \) is given by the sets of its computations, \( C(M) \).

\[ C(M): \omega_1, \omega_2, \omega_3, \ldots, \omega_n, \ldots \]

2.3.2 Statecharts

Statecharts was introduced by D. Harel [Harel 1987] [Cohen 1996] as a visual specification language for specifying complex discrete event entities, including real-time reactive systems. It provides a way to represent state transition diagrams with notions like hierarchy, concurrency, broadcast communication and temporised state. Statecharts is an extension of conventional finite-state machines and their visual counterpart, state-transition diagrams. It describes a system behavior in terms of states, events and conditions, with combinations of the latter two causing transitions between the former [Sowmya 1998].

AND/OR state

There are two types of states in Statecharts which aid in structuring the depth: AND and OR. An OR-state consists of a number of substates; being in the OR-state means being in exactly one of its substates. An AND-state also comprises substates; being in an AND-state implies being in all its substates simultaneously; no transitions are allowed between the substates of an AND-state. Since entering an AND-state means entering every orthogonal component of the state, orthogonality captures concurrency. Orthogonality is the feature the statecharts employ to solve the state blow-up problem, by making it possible to describe
independent and concurrent state components. Also, orthogonality state
decomposition eliminates the need for multiple control activities within a single
activity.

Transition

Transitions in a statechart are not level-restricted and can lead from a state on
any level of clustering to any other. The general syntax of an expression labeling
a transition in a statechart is

\[ a [C] \mid b \]

where \( a \) is a Boolean combination of atomic events (defined later) that triggers the
transition; \( C \) is a Boolean combination of conditions on the events that guards the
transition from being taken unless it is true when \( a \) occurs; \( b \) is an action that is
carried out if, and precisely when, the transition is taken, and this action is also a
Boolean combination of events which will be generated as a result of taking the
transition. All of these are optional. Events and conditions can be considered as
inputs, and actions as outputs. The transition causes \( b \) to occur, which can be
associated with (entering, exiting, or simply being in) a state, in line with the
finite-state machines. In either case \( b \) can be an external event or an internal one,
in the latter case it may trigger other transitions elsewhere in some orthogonal
state.

Events and broadcasting

Atomic events are those events generated by the environment in which the
system functions or those generated within the system itself. Every occurrence of
any event is assumed to be broadcasted throughout the system instantaneously.
Entering and exiting states as well as a timeout, defined by \( tm(e, n) = n \) units of
time since occurrence of event \( e \), are considered to be events. Broadcasting as a
mechanism ensures that an event is made available at its time of occurrence at a
site (state) requiring it, without making any assumptions about the
implementation, and possibly triggering new transitions in other components, and
in general giving rise to a whole chain of transitions.
Chapter 2. Overview of Reactive Systems

Synchronization

Statecharts is said to be synchronous because the system reacts to events by instantly updating its internal state and producing actions, which can trigger in the same instant other transitions. This is named chain reaction causing a set of transitions; the system is always in a waiting state until the condition for a transition is true. By this, the entire chain of transitions takes place simultaneously in one time step.

2.3.3 Temporal Logic

Temporal logic is formalism for specifying structures of states [Kroeger 1986] [Pnueli 1997]. There is a general consensus that every individual run of a program yields a computation which is a linear sequence of states and associated events. We assume that the behavior of a program $P$ is given by $C(P)$, the set of its computations. A predicate $\varphi$ over computations is defined to be valid for the program $P$ if each computation in $C(P)$ satisfies the predicate $\varphi$. In this case we may view $\varphi$ as a valid property of the program $P$, and it can be properly described in the linear time temporal logic [Pnueli 1986].

Temporal logic has been widely used for the specification and verification of concurrent systems [Manna 1981] [Owicki 1980]. It is a linear time logic where attention is on infinite sequences of states of the systems (i.e. behaviors), and their properties. An infinite number of program variables is assumed and every variable in each state of a behavior has a value. In program logic, an assertion or statement (e.g., a relationship between program variables) is true or false when evaluated on a particular state. In temporal logic, an assertion is true or false when evaluated on a particular execution sequence (i.e. a sequence of program states), which plays the role of “time”. As this kind of assertions include a time factor, they can be used to describe interesting temporal properties of programs, or in other words, they can be used to specify assertions about program behavior as time progresses [Leonard 1995].

A program, in general, has two basic properties: safety and liveness, which are
time dependent properties. A safety property states that all finite prefixes of a
computation satisfy some requirement. If the computation is finite, then the
requirement should also be satisfied by the entire computation. A liveness
property complements the safety property by requiring that certain finite prefix
properties hold at least once, infinitely many times, or continuously from a certain
point on. Temporal logic can be used to describe safety and liveness properties,
and it provides two operators on assertions to express these properties: □ (always)
and ◯ (eventually).

Generally speaking, safety means an invariance property; liveness means an
eventuality property. Program invariance is conveniently expressed in temporal
logic terms. An assertion I, is a program invariant if

\[ I \Rightarrow □ I \]

Whereas eventuality can be expressed as:

\[ ◯ I, ◯ □ I \text{ or } □ ◯ I \]

where the safety property □ I states that condition I holds throughout the
computation of the system, i.e., I is an invariant of the system; the liveness
property ◯ I states that condition I eventually holds for the computation of the
system. For example, a traffic light controller needs to assert:

\[ □ ((\text{RedLightOn} \rightarrow ◯ \text{GreenLightOn}) \&\& \text{(GreenLightOn} \rightarrow ◯ \text{RedLightOn})) \]

This is a safety property which includes two liveness properties. The first
liveness property means that there eventually exists a time in the future where
lights will be green, and the second is that there eventually exists a time where
lights will be red. The safety property means that, this will always happen, if the
lights are red then there exists a time further in the future where lights will be
green, and if lights are green then there exists a time in the future where lights
will be red.

Intuitively, safety properties can be characterized as stating "nothing bad will
Chapter 2. Overview of Reactive Systems

ever happen", i.e., a condition \( P_{\text{bad}} \) that is guaranteed never to occur is expressed as a safety property:

\[
\square \neg P_{\text{bad}}
\]

where \( \neg \) is the "No" operator. Liveness properties can be said as that "something good will eventually happen", i.e., a condition \( Q_{\text{good}} \) that is guaranteed to eventually happen is expressed as a liveness property:

\[
\Diamond Q_{\text{good}}
\]

After designing and implementing of a system, it is essential to prove or verify the system's safety and liveness properties. The specification and verification of temporal properties are presented in many literatures (partly see [Manna 1981] [Manna 1983] [Owicki 1980]).

2.4 Reactive Programming

Besides various specification methods, there have been a number of languages proposed to specify and program reactive systems directly [Boniol 1993]. They can mainly be classified as asynchronous and synchronous approaches.

2.4.1 The Asynchronous Approach

Several asynchronous solutions, such as Petri Nets [Genrich 1981], CSP [Hoare 1978] or CCS [Milner 1990], etc. have been proposed to deal with the problem of reactive programming. They are interesting in that they permit hierarchical, modular and concurrent program developments. However, because of the asynchronous nature of task units, they are non-deterministic. Although a communication is seen as a synchronization between two processes, the time taken between the possibility of communication and its actual achievement can be arbitrary. Furthermore, when several communications can take place, their actual order is also arbitrary. Then semantics of the time-handling primitives of these languages is non-deterministic and somewhat vague. Execution times are
Chapter 2. Overview of Reactive Systems

unpredictable, implying difficulties in proving correctness properties. More generally, asynchronous techniques force users to choose between determinism and concurrency, since they base concurrency on asynchronous implementation models, where processes compete for computing resources non-deterministically.

2.4.2 The Synchronous Approach

Synchronous languages have been introduced in the 80's to make the programming of reactive systems easier, such as Esterel [Berry 1988] [Bouali 1996] [Halbwachs 1998] and Statecharts [Harel 1987], etc. They allow reactive systems to be programmed while preserving their natural parallelism and making time reasonings easier. They are based on the simultaneity principle: all parallel activities share the same discrete time scale and then each activity can be dated on this scale. This is called the synchronous hypothesis [Benveniste 1991]: each reaction of a reactive system is assumed to be instantaneous, i.e., takes no time. Such ideal reactive systems produce outputs synchronously with their inputs, their reactions taking no observable time. Then inter-process communication is performed by instantly broadcasting and all sub-processes share the same environment. Synchronous hypothesis provides a deterministic semantics of concurrence, and a formal straightforward interpretation of temporal statement.

However, in order to guarantee as much as possible the synchronous hypothesis, synchronous programs are compiled into deterministic sequential automata. This yields excellent run-time efficiency and predictability. Performance is often as good as that of carefully hand-written code. Nevertheless, such results are not available in the area of distributed reactive programming. Reactive systems we are interested in are potentially loosely coupled and then cannot be implemented by a single sequential automaton. The main problem of the execution of synchronous programs is the reactivity constraint: a synchronous program implements a reactive system; it must thus remain reactive to its environment, in other words, it must react to any of the environment promptings.
Chapter 2. Overview of Reactive Systems

2.4.3 Asynchronous-Synchronous Coupling: Esterel + CSP.

A third approach based on a coupling between the two approaches above has been proposed by G. Berry [Berry 1993]. The general idea is to describe a distributed reactive system as a network of communicating reactive kernels. Each kernel is a synchronous program (Esterel), while communication rules between kernels follow the CSP [Hoare 1978] style, i.e. are based on the “one to one” rendezvous paradigm. Nevertheless, because of the asynchronous semantics of communication and concurrency in CSP, the Esterel + CSP approach yields again non-deterministic semantics, and then non-predictable temporal behaviors for the whole system.

2.4.4 Distribution Method

Many reactive systems have to be distributed on several computing locations for various reasons, including performance increase, location of sensors and actuators, fault tolerance etc. [Caspi 1995]. To achieve a distributed implementation of reactive systems, [Caspi 1999] has proposed a method called the object code (OC) distribution method. This method consists of building the centralized transition system corresponding to the program behavior, debugging and verifying it, and then distributing it according to the system designer’s specifications. This yields the following advantages: first, compiling the program into a single transition system may be useful, for debugging and verification purposes. Second, research on synchronous language compilation has led to a common encoding format for automata: it is the OC format (OC standing for “object code” [Paris 1993]). Hence the OC distribution method can be applied to any synchronous program.

2.5 Related Work -- Tools for the
Development of Reactive Systems

Based on the above specification methods, there have been a number of tools and systems designed to develop complex reactive systems. Many of them use Statecharts or temporal logic to specify the systems and adopt a stepwise approach in general. We review some of them in this section.

2.5.1 STATEMATE

STATEMATE [Harel 1990] [Harel 1998] is a set of tools, with a heavy graphical orientation, intended for the specification, analysis, design and documentation of large and complex reactive systems, such as real-time embedded systems, control and communication systems, and interactive software or hardware. It enables a user to prepare, analyze and debug diagrammatic descriptions of the system under development from three interrelated points of view, capturing *structure, functionality and behavior*. These views are represented by three graphical languages: module-charts, activity-charts and state-charts, respectively. The most intricate of them is Statecharts that is used to depict reactive behavior over time. In addition to the use of statecharts, the main novelty of STATEMATE is in the fact that it “understands” the entire descriptions perfectly, to the point of being able to analyze them for crucial dynamic properties, to carry out rigorous executions and simulations of the described system, and to create running code automatically.

2.5.2 An Object-Oriented Method -- DisCo

DisCo language [J clUvinen 1990] [Systa 1991] [Kellomaki 1997] is a set of tool for specification of reactive systems using the object-oriented method. It uses class and object concepts, and focuses on collective behavior, i.e., how objects cooperate. In DisCo, objects always belong to a class that defines the data and state structures of objects within that class. An object can be understood to consist of two interrelated parts: a finite state part that is a state automation, and a data part
that contains variables and constants. The finite-state structure of a DisCo object is similar to the hierarchical structure used in Statecharts.

The basis of the object-oriented specification is the joint action theory by [Back 1984] [Back 1988]. Joint actions enable us to concentrate on interaction between different components instead of the behavior of individual objects in isolation. An action system can be regarded as an abstract machine consisting of a set of state variables, and a set of actions, where each action consists of a guard and a body. A reactive system can be described as an action system in terms of a collection of classes, actions and assertions. The global state of a system is assumed to be partitioned into objects and local variables, and the system state can be modified only in action bodies, which can refer to the participating objects.

The formal specification method in the DisCo is the Temporal Logic of Actions (TLA) [Lamport 1994], which can be seen as the application of temporal logic in action systems. In TLA, actions are “step predicates” which are valued for steps. Expressions are evaluated for behaviors. For example, with temporal logic operators, we have the following relations:

\[ \Diamond E \equiv \neg \Box \neg E \]

\[ A_1 \rightarrow A_2 \equiv \Box (A_1 \Rightarrow \Diamond A_2) \]

where \( E \) refers to an expression, \( A_1 \) and \( A_2 \) are actions, and \( \rightarrow \) means “leads to”. Intuitively, \( A_1 \rightarrow A_2 \) requires that each step \( A_1 \) will always eventually be followed by a step \( A_2 \).

### 2.5.3 PSP Model

An executable specification language, known as PSP, was introduced in [Heping 1996] for fast prototyping parallel reactive systems. Within PSP, a reactive system can be modelled by a set of abstract devices which act and react with each other concurrently within certain cause-effect constraints. The abstract device model consists of five basic types of components: **physical device, sensor device, actuator device, controller device and communication device**, see Figure
2.2. Various equipment or parallel processing configurations can also be included.

A physical device can be modelled by a function implementing its physical law and affected directly by related actuator devices. An actuator device can be prototyped as a Boolean function or Boolean statement in a physical function to switch on/off a certain action of the physical device. A sensor device can be prototyped to sense a physical device function and report its behavior concurrently to a controller. A controller is an abstract device denoting embedded computer software. It is modelled as a set of programs expressing requested reactive relations between sensors and actuators. It collects assertions from sensor devices or values from the controlled physical devices or communication channels, computes, and then outputs certain assertions and commands to actuator devices to constrain the controlled devices.

![Diagram of a device model for reactive system](image)

**Figure 2.2:** A device model for reactive system

Environmental events can also be modelled by a sequence of inputs from outside of the system and indexed by the user defined global clock. Perhaps an abstract clock device can be added and modelled by functions within the parallel assignment that increment state variables. Running the parallel assignment sequentially, all the above devices may act and react with each other concurrently according to the specified requirements. User's temporal requirements over the system are programmed by state predicates in PSP over the system history. Time is modelled by a set of discrete time points, we call them \textit{timing ticks}.

State variables in PSP are global variables recording the values of system devices over a global clock. They represent the observable behavior of a system.
Chapter 2. Overview of Reactive Systems

State predicates are first order predicates over the state history. To express temporal properties, PSP uses state predicates which refer to the current or previous values of state variables. For instance, in a PSP program

\[ \forall i \in \{0..k\}, \text{pred}(P(i) \text{ var}) \]

where \( \text{pred} \) is a first order predicate, asserts that \( \text{pred}(\text{var}) \) holds for the recent \( k \) steps or \( k \) time ticks. \( P(i)\text{var} \) is the operator used to obtain the value of a state variable at the \( i \)-th previous time tick.

Two temporal operators are defined and implemented in PSP to express safety and liveness properties. They are \textit{always} and \textit{sometime}:

\[ \text{always}(\text{pred}(\text{var})) \equiv \forall i \in \{0..T\}, \text{pred}(P(i) \text{ var}) \]

\[ \text{sometime}(\text{pred}(\text{var})) \equiv \exists i \in \{0..T\}, \text{pred}(P(i) \text{ var}) \]

The first says that \( \text{pred} \) is true for all ticks in the past and the second says that \( \text{pred} \) is true for some ticks in the past.

2.5.4 Structured Reactive System Notation

With the advent of comprehensive safety standards for software intensive safety-related systems, there is a need to establish combinations of techniques which can be used by industry to demonstrate conformance to these standards for particular developments. [Lano 2000] describes one such combination of techniques, defining strategies for controller decomposition which allow safety invariants to be distributed into subcontroller requirements, and techniques for the automatic synthesis of controllers from invariants.
Chapter 2. Overview of Reactive Systems

Figure 2.3: The structure of reactive control systems

In this approach, a reactive control system can be depicted in Figure 2.3, where input event flows are indicated by dashed lines and output command flows by solid lines. This structure corresponds to the convention used on finite state machines. There may be a set of invariants (such as safety and operational behavior properties) associated with each process. [Lano 2000] has proposed the restricted statechart notation to specify reactive systems. The behavior of sensors, controllers and actuators can be specified by Structured Reactive System (SRS) modules in restricted statecharts.

In restricted statecharts all systems are described in terms of modules: an OR state containing only basic and OR states. A system description $S$ is given by the AND composition $M_1 \mid \ldots \mid M_m$ of all the modules contained in it, i.e.,

$$modules(S) = \{ M_1, \ldots, M_m \}$$

Each module $M$ has a set $receiver_{s}(M)$ of modules in $S$, which are the only modules of $S$ to which $M$ can send events.

Typically modules in a reactive system description represent sensors, controllers, subcontrollers and actuators. The invariants for controller behavior, both operational and safety, can be used to synthesize the required control algorithm, i.e., the reaction to individual events.

2.5.5 Agent-based Reactive Systems

There has been a method proposed recently using software agents to specify
Chapter 2. Overview of Reactive Systems

and model reactive systems [Attoui 1995] [Bounabat 1999]. This approach considers a reactive system as a reactive multi-agent system, i.e., a distributed computing system consisting of concurrent reactive agents that cooperate with each other to achieve the desired functionality. Concurrency is characterized by the need to express communication and synchronization among concurrent agents.

Agents are classified as either deliberative or reactive [Ferber 1997] [Nwana 1996]. Deliberative agents derive from the deliberative thinking paradigm: the agents possess an internal symbolic, reasoning model and they engage in planning and negotiation in order to achieve coordination with other agents. Reactive agents don't have any internal symbolic models of their environment, and they act using a stimulus/response type of behavior by responding to the present state of the environment in which they are embedded.

In [Bounabat 1999], a reactive system is defined by a set of agents, connected to each other by communication interfaces. The internal organization of a reactive system consists in a tree, that is made up in parallel of a supervisor (supervisory agent), of two or several sub-agents components, and two communication interfaces between the supervisor and the sub-agents, as shown in Figure 2.4. The system interacts with its environment by the means of actions that is exerted by this environment and external states that is emitted to the environment.

![Diagram of a reactive system structure](image)

Figure 2.4: The structure of an agent-based reactive system

The RDA-based specification of a reactive system ensures that the elements of the system will have time periods coherent with the decision made by the agent,
Chapter 2. Overview of Reactive Systems

and coherent with the periods of decisions made at lower levels of the hierarchy. The temporal constraints are checked on each hierarchy level.

2.6 Summary

It is impossible to develop a complex system without a clear understanding of the system structure and its intended behavior. This chapter first introduced what a reactive system is and what characteristics it has. Then we investigated the specification methods and the various languages and tools for the development of reactive systems.

A reactive system is one that is supposed to maintain an ongoing interaction with its environment, rather than to produce some final result on termination. It mainly has three characteristics: concurrency, determinism, and temporal properties. Most reactive systems are highly concurrent and distributed systems. They are generally composed of concurrent communicating and executing components or sub-processes, either for geographical requirements, or for safety and fault tolerance criteria. Concurrency gives rise to problems that are quite different from the ones sequential systems present, and they form the most difficult tasks faced in the development of reactive systems.

The specification of reactive system behaviors is a very difficult task because such a description may involve complex sequences of those external input and output events, actions, conditions, and information flow, often with explicit timing constraints. The widely used specification methods are Statecharts and temporal logic. The most current tools for the development of reactive systems use these two methods to specify and verify the systems. They have showed that the structure of reactive systems consists of sensors, controllers and actuators.
Chapter 2. Overview of Reactive Systems
Chapter 3

Modeling and Design

Chapter 2 has investigated the reactive system concepts and the current trends for the development of such systems. In this chapter we will propose a generic reactive system architecture based on these concepts, and then build and design the formal model for developing the fault-tolerant computing systems. The model consists of a number of components, such as controllers, sensors, actuators, and application objects, thus we can call it as a component-based architecture model. To formally model this architecture, we have to find an appropriate approach to model reactive components. As we mentioned earlier, a reactive system can be viewed as a reactor that continuously interacts with its environment, we can regard each reactive component as an agent. An actor represents an active object that has a clear defined purpose thus it can be used to model reactive components. Nowadays agent is a very popular concept that is used to model intelligent components [Nwana 1996]. Since a DMM in a reactive system has certain logics or intelligences to some extent, it cannot be regarded only as an actor, rather as an agent. Therefore, in this chapter we try to use agents and actors together to model reactive system components. We will model a reactive system as a distributed computing system consisting of many agents and actors.
Chapter 3. Modeling and Design

3.1 The Generic Reactive System Architecture

So far we have introduced the reactive system concepts and investigated various languages and tools for the development of such systems. Common to all these concepts and tools is that a reactive system mainly focuses on the interactions between the system and its environment; it receives information from the environment (using sensors) and then makes decisions (with controllers) to produce outputs to react to the environment (using actuators); usually, a reactive system is composed of a number of components, such as controllers, sensors, actuators, and physical objects in its environment, which may run in parallel [Boasson 1998]. The tools presented above are extremely useful. They have different focuses on the development of reactive systems and produce different products. However, most of them focus on process control and lack of modularity. They mainly stress on controllers' behaviors and have no emphasis on sensors and actuators. If the control algorithms for a system are changed, the whole system has to be changed. Thus, these methods cannot provide the flexible system architecture for the purpose of fault-tolerance.

We propose a generic system architecture, which can be used in the fault-tolerant computing, from the above reactive system concepts. Our proposal comes from the following method. From the system architecture point of view, it is often convenient to consider real-time systems as composed of three layers. The first layer is an interface with the environment that is responsible for input reception and output production. It transforms external physical signals into internal logical ones. The second one is a reactive sub-system that contains the logical and temporal control of the system. It handles the logical inputs and decides, with respect to the current time, what outputs and what actions must be generated in reacting to the environment. The last one is a set of transformational tasks that perform classical computations requested by the reactive sub-system [Kurki-Suonio 1993] [Edwards 1997].
Chapter 3. Modeling and Design

If we consider a reactive system as composed of three layers, like a real-time system, we can see that the first layer is composed of sensors and actuators. Sensors are used to catch input information or application events from its environment. Actuators are used to carry out control decisions (output) to change its environment or applications. The second layer includes all controllers. Controllers are the core components in the system. They are a set of programs expressing requested reactive relations between sensors and actuators. The main role of the controllers is to make decisions according to the information received from sensors and then use actuators to control or change the environment. Therefore, we can call these controllers as decision-making managers. The last layer is composed of application objects such as physical devices in its environment. These three layers correspond to software process layers: policy, mechanism, and application, i.e., the first layer implements mechanisms, the second implements policies and the third implements applications.

More generally, a reactive system uses sensors and actuators to implement the mechanisms that interact with its environment or applications. Its controllers, we can call them decision-making managers (DMMs), are used to implement the policies regarding to the control of the applications. Therefore we can abstract the generic reactive system architecture, as depicted in Figure 3.1, where DMMs represent the decision-making managers; sensors and actuators connect with the DMMs and application objects by receiving inputs or sending outputs of the system [Zhou 1999].

In this model, a DMM subscribes to sensors and receives reports from these sensors on the applications' 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 monitor some events about the applications). These states or events are sent to the DMMs which react to them by using actuators to change the states of applications.

This architecture is very generic and it can cover all existing reactive systems. For example, in the previous section we investigated that any reactive system is
composed of various controllers, or agents, sensors, actuators and physical application objects. These controllers or agents are categorised as DMMs. Sensors and actuators correspond to the software entities in the architecture. Therefore, any specific reactive systems can be derived from this model and this architecture includes all specific system architectures.

![Generic reactive system architecture diagram]

Figure 3.1: The generic reactive system architecture

The existing reactive systems have a single layer (they mainly focus on process control), thus they lack of the flexibility. Our proposed architecture has a great advantage that it divides the system into three layers and separates policies from mechanisms, i.e., if a policy is changed it may have no impact on related mechanisms and vice versa [Chen 2000(b)]. 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 fault-tolerant computing since it can separate the fault-tolerant computing polices from the fault-tolerant mechanisms. We will elaborate this with detail in the following chapters.

### 3.2 Some Concepts

Since most literatures on reactive systems focus on process control, most concepts about the reactive systems are originated from this area. To develop
fault-tolerant applications, we have to translate the concepts from process control applications into the fault-tolerant computing environment. In particular, we have to understand the following concepts:

- **Fault-tolerant policy** -- The reactive system model has three levels: *policies, mechanisms* and *applications*. 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 [Zhou 1999]. These policies are implemented through DMMs (decision making managers). A DMM obtains information about applications via sensors, processes the information and modifies the states of applications via actuators in order to tolerate component failures.

- **Fault-tolerant mechanism** -- 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. More specifically, it uses sensors to monitor and report some aspects of an application's state. System entities (e.g., DMMs or other sensors) can subscribe to a sensor to receive its report. Actuators in this level provide ways for the DMMs to control the behaviors of applications in order to tolerate component failures.

- **Application** -- The application level deals with issues about fault-tolerant computing application objects, such as database servers, replicas, network routers, etc.

- **Event** -- In fault-tolerant computing, we say an event in terms of failure occurring, recovering, state changing, and object triggering, etc [Hecht 1986]. An event is generated whenever a message is sent by an object through its interface. In addition, we say that an event occurs when a message is received by an object. That is, sending and receiving messages are the only mechanism
for generating events. In a reactive system, input and output are events, the message passing among the components is an event.

- **Failure** – Various failures have been introduced in Chapter 1. A failure occurs when a process, a server, a site, a link fails, or a network partitions [Laprie 1990]. A process failure is the minimum failure but it may lead to partial or total system crash. A server or a site failure could lead to complete processes failure. A link failure or a network partition could lead to different groups in the network fail to communicate with each other.

- **Distributed object** – A distributed system usually consists of many objects distributed all over networks. These distributed objects communicate and collaborate with each other through the networks to achieve the overall system behavior [Daniel 1993]. Many of these objects are likely to fail eventually, causing serious impact on applications, therefore it is essential for the control part of the system to know the current states of these objects.

- **Decision making manager (DMM)** – DMMs are software entities used to implement the policies of a reactive system regarding to the control of applications. They are the control part of the system. A DMM receives reports from sensors on the application states or some events about the applications and then makes decisions according to these reports. It then uses actuators to adjust or change the states of applications. DMMs are distributed objects and may fail [Liu 1994].

- **Sensor** – Sensors are a sort of software entities. They are subscribed by DMMs and responsible for sensing application states or environment events and then reporting to DMMs. Sensors and actuators are together used to implement the mechanisms of the system, or the interface between the system and its environment [Zhou 1998].

- **Actuator (Effector)** – Software entities. Their main role is to change application states according to the decisions made by DMMs.
3.3 Actor and Agent Model

We want to formally build the reactive system architecture model, which we proposed earlier, for the purpose of fault-tolerant computing. This model is a component-based model thus we have to use a method with more modularity to build it. This section introduces actor and agent concepts to model reactive components. By doing so, we can model a reactive system with highly modularity and characterize its concurrency.

3.3.1 Distributed Object Methodology

Chapter 2 has addressed some existing methods for modeling and design reactive systems. They are extremely useful and most of them use statecharts or temporal logic or extended statecharts as the specification languages. But most of them lack of granularity and modularity, and have little discussion on the internal organization of the system. Since most reactive systems are highly concurrent and distributed, the modeling approach should characterize their concurrent components or objects, and their distribution feature as well. Therefore the methods presented in Chapter 2 cannot be used for our purpose of component modeling.

Distributed object methodology can satisfy our purpose. It can be seen as a convergence of the object-oriented methodology and the distributed system technology. By deploying the object-oriented design principles in developing distributed reactive systems, the complexity of constructing and maintaining distributed systems can be greatly reduced [Aarsten 1996] [Barcio 1995] [Kurki-Suonio 1988]. In essence, object-oriented methodology promotes four design principles: modularity, abstraction via encapsulation, object hierarchy via inheritance, and dynamics via polymorphism [Booch 1994] [Meyer 1997]. By enforcing these design principles, incremental design, maintainability, and reusability can be easily achieved. These advantages ascribed to general object-oriented designs hold for distributed object designs.
Distributed objects are extended from classical objects that are independent and working in a networked environment. In contrast to a centralized object system, here is a list of characteristics specific to a system composed of distributed objects.

1. A system of distributed objects spread over networks. The invocation from one object to another often has to across the network.

2. Distributed objects can be heterogeneous, this means objects can be implemented in different languages and run on divergent platforms.

3. Distributed objects fail independently; one object's failure should not bring down other objects of the system. This implies that distributed objects can continue to exist as physical stand-alone entities, and remain accessible to other remote objects running on different machines.

Generally, a classical object is composed of state variables to hold data, and encapsulated by a set of operations to process the data. The state variables can only be accessed exclusively through a set of operations exported by the object. In our system, we use objects to represent all concepts and mechanisms -- system resources, protocols and software entities which constitute a reactive system. Specifically, we will use actor and agent concepts to model the distributed reactive objects.

3.3.2 Actor Model

A reactive system can be viewed as a reactor that consists of a number of actors. Actor concept, originally proposed by [Hewitt 1977] and later developed by [Agha 1986], is the basic concept in the Real-time Object-Oriented Modeling (ROOM) and it can be used to model reactive components [Selic 1992] [Selic 1994]. In general, 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.
Chapter 3. Modeling and Design

The key to identifying an actor is its purpose. Like any object, each actor should have a definite purpose that it fulfills and that justifies its presence in a large system. 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. This shell suggests that the contained functionality is to be viewed as a conceptual unit.

Actors can model functionally significant components in a reactive system, which gives them a fairly broad scope. Significant components occur at many levels in a reactive system starting from the basic units of concurrent execution all the way up to complete systems consisting of multiple layers distributed over a wide physical domain. We model a reactive system as one that is composed of DMM actors, sensor actors and actuator actors.

To communicate with other entities in its environment, an actor provides one or more openings or interface components, which we call ports, in its encapsulation shell. Ports are used for communications between actors, and they allow the exchange of messages between the actor and its environment.

3.3.3 Agent Model

References [Attoui 1995] and [Bounabat 1999] have proposed a method to view a reactive system as a multi-agent system. Bounabat considers that all agents comprising the system are reactive decisional agents based on the decisional concept. This approach works well in terms of modeling decision-making managers, but it does not present sensor and actuator modeling. However, it does give us an idea that we can combine the actor model and agent model together for the modeling of reactive systems, i.e., we model DMMs with agents while sensors and actuators with actors.

In general, an agent is a software entity that can behave autonomously on behalf of its owner. It always lives in a certain environment and can finish its tasks by interacting with the environment [Nwana 1996] [Wooldridge 1995]. An agent possesses the following mandatory properties:
Chapter 3. Modeling and Design

- **Reactive:** senses changes in the environment and acts according to those changes.

- **Autonomous:** has control over its own actions.

- **Goal-driven:** is pro-active.

- **Temporally continuous:** is continuously executing.

To implement the autonomy, each agent has its own execution facility and communication facility by which the agent can interact with the external environment. Such agents are reactive and use sensors and actuators to interact with their environment. They are usually distributed over networks and execute at their own rate and communicate asynchronously by sending and receiving messages.

Agents are concepts originally from Artificial Intelligence (AI) domain. They are knowledge-based, and can model objects with certain intelligence. The most common sense of the agent cited by AI researchers is that the agent is the actor of some event or one who has the power to act, i.e., the intentional agent of some event. In fact, we can say that an agent is an intelligent actor and therefore it possesses all properties of an actor.

### 3.3.4 Messages and Protocols

Communicating with actors in ROOM is based exclusively on message passing. This permits the sender and receiver to be physically separated, making it suitable for distributed system modeling. In this paradigm, the information to be communicated between actors is first transferred to an intermediate message object that is then conveyed from the sender to the receiver by virtue of an underlying communication service.

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.

A protocol class is defined as

- A set of incoming message types
- A set of outgoing message types
- An optional specification of valid message exchange sequences
- An optional specification of the expected quality of service

The terms "incoming" and "outgoing" in this definition imply that a message
protocol specification has a "sideness", or polarity. That is, a protocol is specified
from the perspective of one of the participants. Thus, incoming and outgoing are
defined with respect to the actor to which the interface component is attached.
Two protocols are considered conjugated if they have the same definitions except
that the incoming and outgoing message sets are interchanged. In other words, the
outgoing message set of protocol \( P \) is the incoming message set of conjugated
protocol (denoted as \( P^* \)), and vice versa.

The generic format of a protocol presentation is:

**protocol class** \( P \):

```
  in: \{ (signal_1, data-type_1), (signal_2, data-type_2), ..., (signal_m, data-type_m) \}
  out: \{ (signal_m_1, data-type_m_1), (signal_m_2, data-type_m_2), ..., (signal_m, data-type_m) \}
```

where \( P \) is the protocol class name, and the pairs \( (signal_n, data-type_n) \) specify
individual message types.

Once a protocol class definition has been created, it can be used to define the
actor interface. Interface definition is accomplished by means of ports. A port is a
declaration that the set of messages defined by a protocol class forms as part of
the interface of actors of a particular class. We refer to the interface of an actor, as
defined by its ports, as part of the actor's structure.
3.4 Components Modeling

According to the discussion above, we design a reactive system as an actor system first, which consists of multiple DMM actors, sensor actors and actuator actors. Since a DMM actor has logics or intelligences based on some knowledge, we then view a DMM actor as an agent. Therefore, we formally define that a reactive system is a distributed computing system consisting of multiple DMM agents with sensor/actuator actors as its interface.

3.4.1 DMM Agent

The model of DMM agents is built on the decisional model [Bussmann 1994] allowing the representation of objects according to their behavioral aspects and their degree of intelligence.

Definition 3.1. A DMM agent is a 8-tuple noted <S, D, E, E', P, O, dec, sig>, where

- S: set of signaling received by the agent. Each signaling reflects at any given time the state of the controlled tools or objects used to achieve a specific goal.

- D: set of decisions generated by the agent. Each decision is a solution concerning process behavior in the future; each decision is characterized by its action horizon: Ha, the time during which this decision remains valid.

- E: set of agent's internal states. Each one indicates the current state of the agent.

- E': set of external states set by the agent. Each one represents an object state from the environment.

- P: set of agent's control policies. Each decision is made according to the predefined policies.

- O: set of agent's internal objectives. Each decision is elaborated in order to achieve an internal objective determined by the system policies.

48
The sets above indicate the received events \( S \), the emitted (output) events \( D \), \( E' \), and the internal events \( E, O \).

**Decisional functions.** \( \text{dec} \), \( \text{sig} \) are two decisional functions that define the behavior of a DMM agent.

1. \( \text{dec} \) function.

\[
\text{dec}: S \times P \times E \rightarrow D \times O, \ (s, p, e) \rightarrow (d, o) \quad \text{with} \quad \text{dec}(s, p, e) = (d, o) \Rightarrow \ [s \land p \land e \leftrightarrow d \land o] \quad (3.1)
\]

where \( \rightarrow \) stands for "leads to" and \( \leftrightarrow \) means "simultaneous" (same in the following). This means that depending on a predefined policy \( p \), an appropriate internal state \( e \), and as soon as the receipt of a signaling, corresponding decision \( d \) and the expected internal objective \( o \) are instantaneously produced by the function \( \text{dec} \). Here the signaling is received from sensors about the states of application objects.

In this definition, a decision is caused by a signaling and made by the \( \text{dec} \) function. Obviously, different signalings will cause different decisions, and non-zero signaling will have non-zero decision output.

2. \( \text{sig} \) function.

\[
\text{sig}: O \times S \rightarrow E \times E', \ (o, s) \rightarrow (e, e') \quad \text{with} \quad \text{sig}(o, s) = (e, e') \Rightarrow [o \land s \leftrightarrow e \land e'] \quad (3.2)
\]

This means that depending on the expected internal objective \( o \), and the receipt of a signaling \( s \), its associated external state \( e' \) is instantaneously emitted to the environment, and the new agent internal state becomes \( e \).

**Communication protocols.** The DMM agent has a set of ports to communicate with other objects. The access protocols on the input and output ports are defined as follows:

**protocol class Input**

- \( \text{in:} \ \{\text{signal, Message}, \ \text{error, ErrorCode}\} \)
- \( \text{out:} \ \{\text{enable, Command}, \ \text{disable, Command}\} \)
Chapter 3. Modeling and Design

protocol class Output
  in:  {done, Message}
  out: [{decision, Policy}, {exstate, Message}, {error, ErrorCode}]

where "Input" and "Output" 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. These are same meaning in the following section. Once a protocol class definition has been created, it can be used to define the agent's interface which is accomplished by means of ports.

This definition can be illustrated in Figure 3.2, where signaling is an input event, that is used by dec and sig to generate agent's appropriate responses (decisions and external states); the empty squares represent communication ports; while the squares with names inside represent functions (same in the following figures).

![Diagram of a DMM agent](image)

Figure 3.2: The structure of a DMM agent

**Decision temporal constraint.** The design of reactive systems must take into account the time factor, i.e., temporal constraints [Sahraoui 1994]. The temporal constraint of a DMM can be described as follows. Each decision is characterized by its action horizon, Ha: the time during which this decision remains valid. So, an occurrence of a decision requires the occurrence of its corresponding acknowledgment signaling, in a delay that doesn't exceed its action horizon. This defines the following function, acqDec:

\[
acqDec: D \rightarrow S \times \text{IN}, d \rightarrow (s, Ha) = acqDec(d), \text{ with }
\]

\[
acqDec(d) = (s, Ha) \Rightarrow [d \rightarrow \Diamond_{<Ha, s}]
\]
where

- \( \text{acqDec}(d) \) indicates the acknowledgment signaling of \( d \).
- \( Ha(d) \) is the action horizon of \( d \).
- \( [d \rightarrow \Diamond_{t < Ha} s] = [d \land t = T \rightarrow \Diamond (s \land t \leq T + Ha)] \) means that if \( d \) is true now and the clock reads \( T \) ticks, then within \( T + Ha \) clock ticks, \( s \) must become true.

Let \( C(d) = [d \rightarrow \Diamond_{t < Ha} \text{acqDec}(d)] \), the temporal property that a DMM agent must verify is:

\[
\forall d \in D, d \models C(d) \quad (3.3)
\]

This means that if \( C(d) \) is a formula defining a temporal constraint on a decision \( d \), then \( d \models C(d) \) indicates that \( d \) satisfies the formula \( C(d) \). This constraint guarantees that the decision \( d \) made by the agent must be sent to actuators within \( Ha(d) \) time limit, or it is discarded.

### 3.4.2 Sensor Actor

**Definition 3.2.** A sensor actor is a 7-tuple noted \(<A, E, E', S, O', act, sig>\), where

- \( A \): set of actions exerted on the actor. Each action represents a possible operation to be carried out on an object in order to achieve a special goal.
- \( E \): set of actor's internal states. Each one indicates the current state of the actor.
- \( E' \): set of external states emitted by the actor. Each one represents an object state from the environment.
- \( S \): set of signaling received by the actor. Each signaling reflects at any given time the state of the controlled applications.
- \( O' \): set of actor's external objectives which can be achieved. These objectives represent the actor's interpreting of each action.

**Functions.** The sensor actor has two functions defined as follows.
Chapter 3. Modeling and Design

1. *act* function.

\[
act: A \rightarrow O', \ a \rightarrow o' \quad \text{with} \\
\forall a \in A, \exists o' \in O' \mid o' = act(a) \Rightarrow a \leftrightarrow o'
\] (3.4)

This indicates that the occurrence of an action \(a\) implies instantaneously the occurrence of its associated external objective \(o'\) by the function *act*.

2. *sig* function.

\[
sig: S \times O' \rightarrow E \times E', \ (s, o') \rightarrow (e, e') \quad \text{with} \\
sig(s, o') = (e, e') \Rightarrow [s \wedge o' \leftrightarrow e' \wedge e]
\] (3.5)

This means that depending on a current external objective \(o'\), and as soon as the receipt of a signaling \(s\), its associated external state \(e'\) is instantaneously emitted and the new actor state becomes \(e\). Obviously, different signaling or objectives will produce different external states, and non-zero input will have non-zero output.

**Communication protocols.** The sensor actor has a set of ports to communicate with other objects. The access protocols on the input and output ports are defined by

**protocol class Input**

- in: \{\{action, Action\}, \{exstate, Message\}, \{error, ErrorCode\}\}
- out: \{acknowledgment, Command\}

**protocol class Output**

- in: \{\{enable, Command\}, \{disable, Command\}\}
- out: \{\{signal, Message\}, \{error, ErrorCode\}\}

![Diagram of a sensor actor](image)

Figure 3.3: A sensor actor

52
Chapter 3. Modeling and Design

The structure of a sensor actor is shown in Figure 3.3. Its input events are \((A, S)\) and the emitted events are \((E')\), i.e., the sensor monitors an application's state or action and then reports to DMMs.

**Sensor temporal constraint.** Each external state \(e'\) emitted to the environment is characterized by an acknowledgment signaling \(s\). We define a function, \(acq\), as the acknowledgment of \(e'\):

\[
acq: \quad E' \to S, \quad e' \to s = acq(e') \quad \text{with} \quad \forall e' \in E', \exists s \in S | s = acq(e')
\]

In the sensor actor an occurrence of an external state \(e'\) requires an acknowledgment signaling \(s\), in a delay that does not exceed a maximum time \(M\) defined by users. Let \(C(e') = [e' \to \Diamond_{\leq M} acq(e')]\), the temporal property that a sensor actor must verify is:

\[
\forall e' \in E', e' \models C(e') 
\tag{3.6}
\]

This indicates that once the sensor actor has monitored an external state it must be sent to DMMs within the \(M\) time limit, or it will be discarded.

### 3.4.3 Actuator Actor

**Definition 3.3.** An actuator actor is a 5-tuple noted \(<D, S, E, A, act>\), where

- **\(D\): set of decisions received by the actor.** Each decision is a solution concerning process behavior; each decision is characterized by its action horizon: \(Ha\), the time during which this decision remains valid.
- **\(S\): set of signaling received by the actor.** Each signaling reflects at any given time the state of the controlled applications or tools.
- **\(E\): set of actor's internal states.** Each one indicates the current state of the actor.
- **\(A\): set of actions generated by the actor.** Each action represents a possible operation to be carried out on an object in order to achieve a special goal.
Chapter 3. Modeling and Design

Function. The function act is defined as follow,

\[ act: D \times E \rightarrow A, \ d \times e \rightarrow a \quad \text{with} \]

\[ a = act(d, e) \Rightarrow [a \leftrightarrow d \land e] \quad (3.7) \]

This means that depending on the current decision \( d \), and the internal state \( e \), the associated action \( a \) is produced instantaneously by the function act. Obviously, different decisions will produce different actions, and non-zero decision will have non-zero action output.

Communication protocols. The access protocols on the actuator ports are defined by

protocol class Input
\[
\text{In: } \{\text{decision, Policy}, \text{error, ErrorCode}\}
\]
\[
\text{out: } \{\text{acknowledgment, Message}\}
\]

protocol class Output
\[
\text{In: } \{\text{none}\}
\]
\[
\text{out: } \{\text{action, Action}, \text{error, ErrorCode}\}
\]

The structure of an actuator actor is shown in Figure 3.4. The received events are \( (D) \), which means that the actuator will receive decisions from DMMs, and the emitted events are \( (A) \) meaning that the actuator takes actions according to the decisions.

\[ \text{Decision} \quad \xrightarrow{\text{act}} \quad \text{Action} \]
\[ \text{(Signaling)} \]

Figure 3.4: An actuator actor

Action temporal constraint. An occurrence of an action \( a \) requires the occurrence of its corresponding acknowledgment signaling \( s \), in a delay that doesn't exceed its horizon, \( Ha \): the time during which it remains valid. We define a function, \( acqAct \), as the acknowledgment signaling of \( a \):

\[ acqAct: A \rightarrow S \times IN, \ a \rightarrow (s, Ha) = acqAct(a) \quad \text{with} \]

54
Chapter 3. Modeling and Design

\[ acqAct(a) = (s, Ha) \Rightarrow [a \rightarrow \Diamond_{\leq H_a} s] \]

Let \( C(a) = [a \rightarrow \Diamond_{\leq H_a} acqAct(a)] \), the temporal property that an actuator actor must verify is:

\[ \forall a \in A, a \models C(a) \]  (3.8)

This constraint means that an action \( a \) performed by the actuator must be executed within the \( Ha \) time tick, or it is discarded.

3.5 Reactive Behaviors

We refer to the internal operation of an actor over time as its behaviour. When an actor is required to operate differently during different time periods, we represent the behaviour changes as caused by changes in state [Selic 1994]. 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 statecharts, which constitute an extensive generalisation of state-transition diagrams.

3.5.1 States and Transitions

Informally, a state is a static condition during which the object is receptive to new events. When an event occurs, the object responds by changing its state to reflect the new history of the object. The new state depends on the event that occurred, as well as on the previous state. This transfer from one state to another is called a state transition. The event that causes a state transition is called a triggering event or, simply, a trigger. When a transition is triggered by an event, the object may perform some action that can generate new events.

One common interpretation of the concept of state for software systems is that each state represents one distinct set of valid values of all program variables, but
Chapter 3. Modeling and Design

this will lead to a very large number of states. So states are abstract views of an object that determine the qualitative aspects of its behavior, while variables (and their values) pertain to its detailed quantitative aspects. These variables are just called extended state variables.

A transition may have an action associated with it. An action consists of a set of action steps that are limited to operating over any of the following types of objects:

- **Extended state variables** that are used to store the messages automatically that caused the current event so that they can be accessed if necessary.

- **Temporary variables** that are used to store inter-mediate results during the execution of an action.

- **The interface objects (ports)** that are accessed to communicate with other actors.

When an actor is created, it does not start off in a state but instead takes an initial transition, which is the start of a new concurrent execution thread and is one of the features that distinguishes an active object from a passive one. The typical activities performed in an initial transition include initialisation of extended state variables and the dispatching of messages for synchronization with other actors.

With the exception of the initial transition, all other activity in an actor is triggered by the arrival of events at one of the interface components. A simple trigger specification consists of the following three elements:

- The **name of the triggering signal** of the event that causes the transition

- The **name of the interface component** on which the triggering event is expected to arrive

- An optional **guard condition**

The triggering signal must be a valid input signal. The guard condition is a Boolean expression that is evaluated dynamically when the event is scheduled for
processing. The trigger is satisfied, and the associated transition taken, if and only if the expected signal arrives at the specified interface component and the guard condition is evaluated to "true".

Summarizing above, a transition can be specified as the following generic textual form:

\[
\text{transition } T:
\]

\[
\text{triggered by: } \{ \text{signal}_1, \text{interface component}_1, \text{guard}_1 \}
\]

\[
\text{or } \{ \text{signal}_2, \text{interface component}_2, \text{guard}_2 \}
\]

\[
\text{or } ...
\]

\[
\text{action: } \{ \text{...action code...} \}
\]

The \text{triggered by} clause is optional since some transitions do not have triggers. The action code may be empty or it may contain detailed code.

The behavioral model of state machine consists of a series of alternating pauses (modeled by states) and actions (modeled by the transitions). A state machine graphical diagram consists of squares with two circles at two ends, which represent states, and arcs that represent transitions.

### 3.5.2 Behaviors of DMM Agents

The scenario of a DMM agent is described as follows. The DMM first subscribes to sensors and receives their reports on application states. The input events of the agent are \( S \), received from sensors about the states of applications and actions. The DMM will make decisions (\text{dec}) according to this information and its intelligence (\( o \)). Its output events, thus, are (\( D, E' \)), i.e., it sends out decisions it made to actuators to change the states of applications and sets up (\text{sig}) the external object state (sensor's new state). The purpose of the DMM agent is to implement a policy according to the information received from various sensors.

According to its functions, we design the behavior of the DMM agent as depicted in Figure 3.5, where the circle with \( I \) is the source of initial action.
During its lifetime, the DMM progresses through three basic phases. On creation, it is initialised and enters "Subscribing" state, during which it subscribes and listens to sensors connecting to it. Once there are some sensors connected to it, the DMM enters the "Waiting" state. In this state, it awaits to receive the sensor reports. If it can receive requests for connection from new sensors at this stage, the DMM will go back to "Subscribing" state again accompanied by the transition "listening". After it receives a signaling from a sensor the DMM will enter the state "Operational". During this state the DMM makes decisions according to the information it received. Then it sends the decisions to actuators and return to the "Waiting" state again. By the transition "sending", the DMM sets up sensors’ new states (enable or disable).

From Figure 3.5, we see that there are five types of transitions in the DMM. The descriptions of the transitions are listed as follows:

- **transition connecting**:
  - **triggered by**: \{request, sensorPort\}  
  - **action**: \{connect()\}

- **transition listening**:
  - **action**: \{listen()\}

- **transition receiving**:
  - **triggered by**: \{signal, sensorPort\}  
  - **action**: \{receive()\}

- **transition decisionMaking**:
  - **triggered by**: \{event, actuatorPort\}  
  - **action**: \{dec()\}

- **transition sending**:

58
triggered by: \{\text{signal, actuatorPort}\} \hspace{1em} \text{action: } \{\text{sig()}\}

where request, signal, and event are input signals; sensorPort and actuatorPort are the input and output ports of the DMM; connect(), listen(), and receive() are functions.

3.5.3 Behaviors of Sensors

The behaviors of sensors and actuators are simple. The function of a sensor is to monitor and report the states of application objects to its subscribers (i.e. DMMs). Its input events are \((A, E')\), for instance, an application object may have an action exerted on the sensor or its state is captured by the sensor. The sensor's emitted or output events are \(S\), i.e., the sensor reports the action information translated by \(act\) and the states of applications to the DMMs.

![State Machine Diagram for Sensor Behavior](image)

Figure 3.6: The behavior of a sensor actor

Figure 3.6 shows the state machine diagram of the behaviour of a sensor actor. During its lifetime, the sensor actor has three basic phases. After initialised, it enters the “Waiting” state, during which it awaits for connection with its subscribers (DMMs). Once it connects with the DMMs, the sensor enters the “Monitoring” state, in which it monitors events in the environment. And also it listens to new subscribers, and will return to the “Waiting” state again if there are new DMMs coming. Once some events happen, the sensor enters the “Operational” state accompanied by the transition “catching”. In Operational, it
translates actions and reports the events to the DMMs in the order in which they occur. After that, it returns to “Monitoring” state again.

There are five transitions in the sensor actor which are listed as follows:

**transition connecting:**  
triggered by: [request, dmmPort]  
action: {connect()}

**transition listening:**  
action: {listen()}

**transition catching:**  
triggered by: [event, appPort]  
action: {catch()}

**transition translation:**  
action: {act()}

**transition reporting:**  
triggered by: [signal, dmmPort]  
action: {sig()}

where request, event and signal are input signals; dmmPort and appPort are the ports of the sensor actor; connect(), listen() and catch() are functions that catch events from the environment and report to DMM.

### 3.5.4 Behaviors of Actuators

The purpose of the actuator actor is to receive decisions made by DMMs, and then to change the states of the relevant application objects according to the decisions. Its received events are \( (D) \), i.e., decisions from DMMs. The emitted or output events are \( (A) \) that are actions the actuator takes according to the decisions.

The behavior of an actuator actor is shown in Figure 3.7. Similarly, the actuator actor has three basic phases: “Listening”, “Waiting” and “Operational”. After initialisation, it enters the “Listening” state in which it builds connections with its subscribers (DMMs). After that, it enters the “Waiting” state, during which it awaits orders from the DMMs. Once a decision arrives, it enters the “Operational” state in which it takes the action on changes of the state of the relevant application object according to the decision, and then returns the “Waiting” state again.

There are four transitions listed as follows:

**transition connecting:**
Chapter 3. Modeling and Design

triggered by: \{request, dmmPort\} \hspace{1cm} action: \{connect()\}
transition receiving:

triggered by: \{event, dmmPort\} \hspace{1cm} action: \{receive()\}
transition actTaking:

triggered by: \{event, appPort\} \hspace{1cm} action: \{act()\}
transition emitting:

Figure 3.7: The behavior of an actuator actor

3.6 System Modeling

Based on the above definitions for DMMs, sensors and actuators, we can give the system definition now, which we design as a distributed computing system consisting of multiple DMM agents and sensor/actuator actors. These agents and actors are connected to each other by communication ports.

Definition 3.4. A reactive system \( R \) is a collection of DMM agents, sensor actors and actuator actors, which are running in parallel, i.e.

\[ R := \{ D \times S \times A \} \]

where

- \( D = \{D_1 \parallel D_2 \parallel ... \parallel D_n\} \), set of DMM agents running in parallel. Let \( R_m(s) \) denote the response of a module \( m \) to an input \( s \), so
Chapter 3. Modeling and Design

\[ R_D(s) = R_{[D][D_2]...[D_m]}(s) = R_{D_1}(s) \parallel R_{D_2}(s) \parallel ... \parallel R_{D_m}(s) \]

We assume that the system $R$ is deterministic, i.e., the coordination between the DMMs guarantees that: if given the same input, $D$ will make the same decisions, i.e.,

\[ R_D(s_1) = R_D(s_2) \text{ if } s_1 = s_2. \text{ } s_1, s_2 \text{ are input signalings.} \]

- $S = \{S_1 \parallel S_2 \parallel ... \parallel S_n\}$, set of sensor actors running in parallel. Each sensor runs independently without impact on other sensors, thus we have

\[ R_S(e) = R_{\{S_1[S_2]...[S_n]\}}(e) = R_{S_1}(e) \parallel R_{S_2}(e) \parallel ... \parallel R_{S_n}(e) \]

\[ R_S(e_i) = R_S(e_2), \text{ if } e_1 = e_2. \text{ } e_1, e_2 \text{ are input events.} \]

- $A = \{A_1 \parallel A_2 \parallel ... \parallel A_k\}$, set of actuator actors running in parallel. Each actuator runs independently without impact on other actuators. We have:

\[ R_A(d) = R_{\{A_1[A_2]...[A_k]\}}(d) = R_{A_1}(d) \parallel R_{A_2}(d) \parallel ... \parallel R_{A_k}(d) \quad \text{and} \]

\[ R_A(d_i) = R_A(d_2) \text{ if } d_1 = d_2. \text{ } d_1, d_2 \text{ are input decisions.} \]

- $R = M(e), \text{ } e \in I. I$ is a set of input events. \( M(e) = R_A(R_D(R_S(e))) \), is a set of output events.

Definition 3.4 indicates that a reactive system ($R$) uses sensors ($S$) to monitor and capture input events from the environment, and uses actuators ($A$) to output its responses, made by DMMs ($D$), to the environment. This definition can be depicted in Figure 3.8, where $I$ is input events about actions and states of application objects in the environment, and $O$ is output events about actions the system responses to the environment.

![Diagram of a reactive system](image)

Figure 3.8: The definition of a reactive system

62
3.6.1 Safety and Liveness

Given its definition, the reactive system has a series of properties that need to be ensured after the design and implementation. As we introduced in Chapter 2, the properties of distributed systems are mainly classified as safety and liveness properties [Pnueli 1986].

3.6.1.1 Safety Property

Safety property is an invariance property of the system. It indicates that something bad never happens. From the fault-tolerant computing point of view, safety means that the system always runs correctly, i.e., fault-tolerance ensures the system safety. Based on this knowledge, we have the following assertions about safety property of reactive systems.

Assertion 3.1. Any input event can be responded by the system R within a certain time (e.g. T); and different input events will lead to different responses, i.e.,

\[ \forall e \in I, \exists r \in O / r = M(e) \Rightarrow [e \rightarrow \Diamond_{<T} r] \] and

\[ \forall e_1, e_2 \in I, \exists r_1, r_2 \in O / e_1 \neq e_2 \Rightarrow r_1 \neq r_2 \]

Proof. According to the temporal constraints (3.3), (3.6), (3.8), we have

\[ \forall e \in I (e \neq 0), \exists s = R(e), \text{ and } s \models [s \rightarrow \Diamond_{<M} \text{acq}(s)], \text{i.e.} \]

s is produced within the M (see 3.6) time ticks. Similarly, the system can generate d within the Ha(d) (see 3.3) ticks and r within the Ha (see 3.8) ticks. Assume \( T = M + Ha(d) + Ha \), the system produces the response r within the T ticks, i.e.,

\[ r = M(e) \Rightarrow [e \rightarrow \Diamond_{<T} r] \]

That means, for each input event e, the system R will respond to it with r within the time T.

For the second clause, according to dec, sig, act from Definition 3.1, 3.2, 3.3, a DMM, a sensor, or an actuator will produce different results when fed
with different inputs. That means the system will generate different responses when it has different inputs.

**Assertion 3.2.** Output events are generated in the same order that their cause events have been input (first-in–first-out discipline).

**Proof.** Suppose \( e_1, e_2 \) are input events, and \( e_1 \neq 0, e_2 \neq 0 \), we have

\[
  r_1 = R_A(R_D(R_S(e_1))) \quad \text{and} \quad r_2 = R_A(R_D(R_S(e_2))).
\]

Let \( t_n \) denote the time tick for \( n \) event, this assertion is equivalent to that if \( t_{e_1} < t_{e_2} \), then \( t_{r_1} < t_{r_2} \). According to the temporal constraints (3.3) (3.6) (3.8), the responses (from each DMM, sensor or actuator) to an input event must be sent within a short period of time (Ha(d), M, Ha) if their responses remain valid. Before they send the current responses, they do not process the next input event. Therefore, if \( t_{r_1} < t_{r_2} \) then \( t_{r_1} < t_{r_2} \); as \( t_{e_2} < t_{e_2} \), so \( t_{e_1} < t_{e_2} \).

**Assertion 3.3.** Same input events will incur the same responses from the system \( R \), i.e.,

\[
  \forall e_1, e_2 \in I, \exists r_1, r_2 \in O, r_1 = M(e_1), r_2 = M(e_2),
\]

if, only if \( e_1 = e_2 \), then \( r_1 = r_2 \).

**Proof.** Suppose \( e_1, e_2 \) are input events, and \( e_1 \neq 0, e_2 \neq 0 \), we have

\[
  r_1 = R_A(R_D(R_S(e_1))) \quad \text{and} \quad r_2 = R_A(R_D(R_S(e_2))).
\]

From Definition 3.4, if \( e_1 = e_2 \), then

\[
  r_2 = R_A(R_D(R_S(e_2))) = R_A(R_D(R_S(e_1))) = r_1
\]

This property is the deterministic characteristic of a reactive system we mentioned earlier, i.e., if fed with same input events, the system would have the same behavior. That means, the system behavior can be predicted.

If all these properties are guaranteed after the system design and implementation, the system will always run correctly, and we say that the system is fault-tolerant.
3.6.1.2 Liveness Property

Liveness property is an eventuality property. It means that something good eventually happens. The nature of the reactive system is its reactivity, therefore it must guarantee that it will always be able to respond to its input events.

Assertion 3.4. Each input event of the reactive system \( R \) will eventually be responded by an output action, i.e.,

\[
\forall e \in I, \exists ! r \in O \mid r = M(e) \Rightarrow \delta(e \leftrightarrow r)
\]

Proof. From Definition 3.2, we have

\[
\forall e \in I, e \neq 0, \exists s = R_s(e) \neq 0.
\]

Similarly, from Definition 3.1, 3.3, we can have

\[
\forall s \in S, s \neq 0, \exists d = R_D(s) \neq 0; \text{ and } \forall d \in D, d \neq 0, \exists a = R_A(d) \neq 0.
\]

Hence, \( \forall e \in I, e \neq 0 \Rightarrow \exists r = a = R_A(R_D(R_s(e))) \neq 0 \). This means that the system \( R \) will generate a response \( r \) for an input \( e \). Also, according to the Assertion 3.1, any input event can be responded by the system \( R \) within a certain time (e.g. \( T \)). Hence any input event of the reactive system will eventually be responded by an output action.

The reactivity is the most important property of a reactive system. To ensure this property, the system must guarantee the temporal constraints for each DMM, sensor and actuator. If they all meet the constraints, i.e., they can send out their outputs within the time limits, the system will eventually have the response emitted.

The most common example of liveness in distributed systems is termination [Gartner 1999]. Thus we have another assertion for the liveness property.

Assertion 3.5. If given a reactive system \( R \), it will eventually be terminated.

Proof. From Assertion 3.1 and Assertion 3.4, we know that if given an input \( e \), \( R \) will eventually respond to it; if no input, \( R \) will produce no output. Thus, when inputs from outside are ended, the system \( R \) terminates.
3.6.2 System Architecture

Besides the safety and liveness properties, we can get another important property on the system architecture derived from the system definition.

**Assertion 3.6.** A reactive system $R$ can remain its reactivity if $S$ (sensors) or $A$ (actuators) have been changed while $D$ (DMMs) retains unchanged, and vice versa.

**Proof.** From **Definition 3.4**, a reactive system consists of DMM agents, sensor actors and actuator actors, each of which performs a specific task. From **Definition 3.1**, we know that each DMM runs according to its input signaling, received from whatever sensors, and sends its decisions to whatever actuators. It is independent from any sensors and actuators. So does a sensor or an actuator. Therefore, in the reactive system, DMMs, sensors and actuators are independent with each other, and any of them has no impact on others if it is changed, as long as the information flow (I/O) in the system stays the same.

This property means that we can separate policies from mechanisms, i.e., we can separate DMMs with sensors and actuators in the system, so that they can be changed respectively without impact on each other. This property is an advantage of the reactive system model we proposed in the thesis.

The structure of the reactive system is depicted in Figure 3.9, where each DMM connects with multiple sensors and actuators by communication ports, some of which with $C$ are the ports with conjugated protocols because the protocols on them are conjugated with those on the ports of sensors and actuators. Each DMM may have a port to connect with a supervisory agent to coordinate the system behavior. The system interacts with its environment by the means of actions that are emitted by actuators and external states that are captured by sensors. In such a system, a DMM may subscribe to multiple sensors and actuators that are attached to different applications respectively, and each sensor/actuator can connect to multiple DMMs as well. The communications
between them are addressed in the next chapter.

![Diagram of system structure]

**Figure 3.9: The reactive system structure**

### 3.7 Summary

In this chapter, we have proposed the generic reactive system architecture and then build the formal model for it. Based on the reactive system concepts investigated in Chapter 2, we abstracted the generic reactive system architectural model. This model consists of three layers: policies, mechanisms, and applications. The policies layer is implemented through decision making managers (DMMs) regarding to the control of the applications, while the mechanisms layer is implemented through sensors/actuators. The major advantage of this architecture is that it separates policies from mechanisms. This advantage leads to a flexible software structure and has a great significance in developing fault-tolerant computing systems since it can separate fault-tolerant computing policies from its mechanisms.

Since this architecture is composed of many components, we introduced the actor and agent concepts to model these reactive components. An actor represents an active object that has a clearly defined purpose and it may have its own execution thread and can operate concurrently with other active objects in its
domain. Therefore, the actors can reflect the reactive nature of components in reactive systems. However, a DMM has some intelligence, and it is more like an agent. An agent is a software entity that can behave autonomously on behalf of its owner. It is reactive, autonomous, and goal-driven. In fact, an agent is an actor with intelligence. To better describe the nature of the DMM behavior, we use agents to model the DMMs in reactive systems. Therefore, we consider that a reactive system is a distributed computing system consisting of multiple DMM agents and sensor/actuator actors. This combination of actor and agent models has a good description of the reactive behaviors and has strong semantics, and increases the system modularity as well.

The behaviors of DMMs, sensors and actuators are described in statecharts. We describe that a reactive system interacts with its environment by the means of actions that are emitted by its actuators and external objects’ states that are captured by its sensors. The communications between DMMs and sensors/actuators are through their ports. Particularly, we defined the temporal constraints for each DMM, sensor and actuator. These constraints form the basis of the whole system. They ensure that a reactive system can remain its reactivity.

The system description is based on the components modeling. A reactive system is a collection of DMM agents, sensor actors and actuator actors which are running in parallel. After giving the system definition, we derived a few system properties from the definitions. The most common properties of distributed systems are safety and liveness properties. These properties are ensured by the system fault-tolerance, i.e., if the system is fault-tolerant, it will always run correctly. Besides these, we also deduce an important property that is the separation of policies and mechanisms from the system architecture. This property is the major advantage of the reactive system model.
Chapter 4

Communication Services

In many cases, there are multiple DMMs, sensors and actuators in a reactive system and their communication with each other becomes much complicated. Usually, a reactive system distributes controllers (DMMs), sensors and actuators across a number of computers, which run in parallel. Different DMMs may need to communicate with each other. The coordination between them remains an important task.

In such a system, a DMM may need to know multiple application's states to make correct decisions for the control of them, and a sensor/actuator performs its sensing/switching function on one application object only, therefore the DMM needs to subscribe to multiple sensors and actuators for the control of multiple application objects. The different arrival ordering of messages (from these sensors) will cause the DMM to make different decisions. For example, in a traffic light control system, different order of vehicle arrival in an intersection will have different effects on the decision making. On the other hand, an application's state change may have an impact on other related application objects, thus there may be multiple DMMs that want to know this application's state. Therefore, a sensor may need to report to multiple DMMs simultaneously. Similarly, an actuator can receive decisions from multiple DMMs but perform its function on one application object only.
Chapter 4. Communication Services

Hence, the communication between these components cannot be described only using the exchange of single messages. It involves group communication services, such as message ordering and multicast services [Birman 1993]. Therefore, the group communication mechanism can be applied in the reactive system model, i.e., we may consider a reactive system as a group communication system.

Group communication systems provide reliable and ordered multicasting primitives for building group oriented distributed systems. They use a multicast message to communicate between a group of processes [Powell 1996] [Liang 1990]. A multicast message is one that is sent by one process to the members of a group of processes. Multicast messages provide a useful infrastructure for providing fault tolerance in distributed applications and they are a very useful tool for constructing distributed systems.

A reactive system is a proper candidate for using group communication primitives, since reports originated at different sensors need to be multicasted to the group of DMMs, and so does the decisions from different DMMs to the group of actuators. Following sections will address the group communication services in a reactive system. Before that we discuss the process scenario first.

4.1 Processes in a Reactive System

The main task of a reactive system is to continuously react to applications in its environment by producing outputs. There are three types of processes in the system: DMM process, sensor process and actuator process. As mentioned before, a DMM process may subscribe to multiple sensor/actuator processes and a sensor/actuator process can report to (or receive decisions from) multiple DMM processes as well. To implement this, we use multi-threaded entities to model DMMs, sensors and actuators, respectively. Therefore, a reactive system can be modelled by the use of processes and threads with following tasks:

- A sensor process continuously captures new information from an application
object and reports it to all DMMs which have registered interests in the corresponding application. There will be a separate sensor process for each application object, and each sensor process possesses multiple threads each of which is responsible for connecting and reporting to a DMM. The sensor is a multi-threaded entity.

- A DMM process registers the interest with relevant sensors and receives reports from them on the states of corresponding applications. It then makes decisions according to these reports and sends the decisions to related actuators. A DMM process may have multiple threads each of which is responsible for the connection with one sensor or actuator process. The DMM is a multi-threaded entity and different DMM processes may have different number of threads.

- An actuator process registers the interest with relevant DMMs. It then receives decisions from them to change the corresponding application’s state. There will be at least one separate actuator process for each DMM and each actuator can have multiple threads each of which is for the connection with one DMM.

The communication between these processes and threads is shown in Figure 4.1. We discuss the communication between sensors and DMMs first. A sensor process sends a report on the state of an application simultaneously to all the DMM processes, which have registered interests in it, using its multiple threads. In order that a whole system containing a group of DMM processes makes a correct response to a particular application, all of the DMM processes, which have interests in this application, should receive a report about the state of this application at the same time from the sensor attached to it. However, in a distributed environment, due to some reason (e.g. a failure occurring) some of DMMs may not be able to receive the report. If there is one (DMM) that cannot receive the report, other DMMs should receive the report and a notification of the report should be sent to the failed DMM after it is recovered. Otherwise, the system may generate some errors because different DMMs may make conflicted decisions.
Chapter 4. Communication Services

Furthermore, these DMM processes may need information from more than one application to make decisions, so they subscribe to a number of relevant sensors using their multiple threads. To ensure that these DMMs have correct information, they should receive the reports from these sensors in the same order. That means, the information flow between these DMMs and sensors should be ordered.

![Diagram of communication in a reactive system](image)

Figure 4.1: Communication in a reactive system. Each oval represents a process; each arrow represents a thread; lines with double arrows mean that DMMs send their decisions to each other. This Figure shows that a DMM process may subscribe to multiple sensor/actuator processes and a sensor/actuator process can report to (or receive decisions from) multiple DMM processes as well.

Similar situation happens in the communication between DMMs and actuators. From these discussion and analysis, we conclude that it is necessary to ensure the ordered and complete communication among DMMs, sensors and actuators in a reactive system.

4.2 Correctness Requirements

According to the communication among the reactive components, we define the basic correctness requirements for a reactive system as:
Chapter 4. Communication Services

- **Completeness.** If a DMM receives a report from a sensor, all other DMMs that subscribe to this sensor should receive it as well. Similarly, all actuators that are appointed by a DMM should receive a decision from the DMM if any one of them receives it. This property is to guarantee that all DMMs receive the same set of reports from a sensor so that they all can have this information to make decisions, and all actuators can receive the same decision from their DMM.

- **Message delivery ordering.** Message delivery among DMMs, sensors and actuators should have ordering constraint. At least the FIFO ordering is the default constraint among them. This is to guarantee that all DMMs get the correct information and applications get the correct responses of their inputs.

  Different ordering constraints affect the performance of the system. A strict ordering constraint brings safety and correctness property to the system, but reduces efficiency, especially the system throughput rate. Therefore, it is necessary to analyse the message set to find an ordering constraint which has the least necessary strength to meet the semantics requirement.

  A message from a sensor can be processed at DMMs right away without being deferred if the message ordering constraint is satisfied. Consequently, the decision is sent to actuators promptly, and this can speed up the flow of message delivery performed by the system to improve the system throughput rate. Messages originated at different sensors can be handled in parallel in different DMMs if their ordering constraints are satisfied, such as total or causal ordering.

  However, in the reactive system group, members are not equally weighted, e.g., DMMs have greater weights than sensors and actuators because they will either receive multicasts from sensors or send multicasts to actuators, whereas sensors or actuators only send or receive multicasts. Hence, there should be different ordering constraints in the group, and we should distinguish the three sub-groups: DMM, sensor and actuator, from the original group, so that we can apply different constraints to them. In the DMM group, each member will either receive or send out multicasts, while the members of the sensor group only send
multicasts and the actuator group only receive multicasts.

In the first chapter, we have mentioned there are a few types of failures which could happen in a distributed system. We assume crash failure [Cristian 1991] semantics in this thesis, i.e., we only discuss the crash failure in a reactive system because this type of failure is the most possible case. The crash failure is defined closely to fail-silent failure addressed in Chapter 1. It has the semantics – a process/processor behaves exactly correct according to the specification until it suddenly halts; after ceasing its activities, there is no more output sent out, and the data stored in the volatile memory is lost, and the state of ceased process is unknown.

The correctness of the reactive system model is subject to the site crashes. A crash on any site of DMM, sensor or actuator may lead to a disastrous result and cause the incomplete communication. Since sensors and actuators are attached to applications, if a sensor or an actuator fails, the application site they are attached to may crash as well. Hence the DMMs, which subscribe to this sensor/actuator, may not receive complete reports from the sensor, i.e., some of them may receive and some may not, thus they may make conflict decisions. If a DMM site crashes, the system may not be able to produce the prompt responses.

4.3 Ordering Constraints

Without a mechanism to ensure ordered delivery of messages, when two originators multicast to a group at about the same time, their messages may not arrive in the same relative order at members of the group. For instance, this may happen if one of the messages is dropped by one of the recipients and has to be retransmitted.

Considering a reactive components group, when different messages are propagated between the group members concurrently, their arriving orders may be different. Figure 4.2 depicts this scenario. This scenario is the result of different network latencies on communication links between the members and different
speeds of machines on which the group members are running.

To ensure the correct semantics of the reactive system, a sensible arriving order of message delivery has to be assigned and enforced over the whole reactive components group. According to the group communication mechanism, there are three types of ordering: *FIFO ordering, totally ordering and causal ordering.*

Figure 4.2: Message delivery in a reactive system – Two reports \( r_1, r_2 \) from a sensor are sent to two DMMs which produce three decisions \( d_1, d_2, d_3 \) that are sent to an actuator.

### 4.3.1 Ordering Definitions

**Definition 4.1:** *FIFO ordering constraint.* If two messages \( m_1 \) and \( m_2 \) originated from the same component are sent to the components group, and if \( m_1 \) is originated before \( m_2 \) at the original component, then \( m_1 \) will be delivered before \( m_2 \) at the rest of components in the group, which they are delivered to.

FIFO is the constraint defined between one sender and a set of receivers. It requires requests from the same sender to be delivered First-In-First-Out at all receivers. In the reactive system model, this ordering is understood as reports sent by the same sensor are to be processed in DMMs in the order they are sent, and so does decisions sent by the same DMM in actuators. This ordering is simple to achieve by attaching sequence numbers to messages.
Chapter 4. Communication Services

In Figure 4.2, r1, r2 and d1, d3 originate from the same components (sensor and DMM1) respectively, and d1, d3 arrive at the actuator meeting the FIFO constraint while r1, r2 arrive at DMM1 and DMM2 violating the FIFO constraint.

**Definition 4.2:** Total ordering constraint. For two messages \( m_1 \) and \( m_2 \) (originated from the same or two different group members), if any group member delivers \( m_1 \) before \( m_2 \), the rest of members deliver \( m_1 \) before \( m_2 \) as well, as long as both \( m_1 \) and \( m_2 \) can reach them. Or, the other way around, if one member delivers \( m_2 \) before \( m_1 \), the rest of members deliver \( m_2 \) before \( m_1 \) as well.

Total ordering means that when several messages are transmitted to a group the messages reach all of the members of the group in the same order [Powell 1996]. It requires \( m_1 \) and \( m_2 \) to be delivered either in the order of \((m_1, m_2)\) or \((m_2, m_1)\), as long as the ordering is consistent at all members. In the reactive system, total ordering means that all messages from different sensors are delivered to every DMM in the same order, or all decisions from different DMMs are delivered to every actuator in the same order. In Figure 4.2, neither of ordering on DMM members complies by the total ordering. If the total ordering is required, any ordering of \((r1, d1, r2, d3, d2)\) or \((d1, r2, r1, d2, d3)\) delivered on all members is fine.

Total ordering applies to all the messages sent to a group from all senders. Total ordering is potentially expensive in its use of communication. Causal ordering is an ordering property that is less restrictive than total ordering but meets most application requirements. It is based on the notion of causality.

**Definition 4.3:** Causal ordering constraint. If a message \( m_2 \) originated from \( G_j \) is caused by a message \( m_1 \) originated from \( G_i \) (\( G_i \) and \( G_j \) are group members), then \( m_1 \) is delivered before \( m_2 \) at the rest of members which both \( m_1 \) and \( m_2 \) are delivered to.

Causal ordering generalises two relationships:

- If two events occurred at the same process, then they occurred in the order in which it observes them.

76
Whenever a message is sent between processes, the event of sending the message occurred before the event of receiving the message.

Causal ordering is a natural extension to FIFO by considering different senders. It came from happened-before relation defined in [Lamport 1978]. The happened-before relation defines that if \( m_i \) causally (happened-before) precedes \( m_j \), \( m_i \) should be delivered before \( m_j \) at all common receivers of \( m_i \) and \( m_j \). Causal ordering is understood as that, if two messages have the nature of cause-effect relation, this relation should be kept at all members. In the reactive system, any decision will have cause-effect relations with one or more messages received from sensors before.

In Figure 4.2, \( r_2 \) and \( d_3 \) is satisfied at DMM2. However, \( d_1 \) arrives earlier than \( r_1 \) at DMM2. This violates the causality constraint, thus, \( d_1 \) has to be deferred until \( r_1 \) is delivered.

### 4.3.2 Assigning Ordering Constraints

According to three sub-groups in a reactive system, we can categorize the following types of message sets and assign them with different ordering constraints respectively.

- Firstly, messages originated from each specific component, such as each sensor or DMM, forms a message set. For this message set, we should assign the FIFO ordering constraint on it, for they are simple and originated from one sender, respectively. As we see, the sensor group does not receive any message thus we should check the messages that are originated from the same sender with the FIFO constraint in the DMM and actuator groups.

- Secondly, each DMM (or each actuator) will receive a number of messages from different sensors (or DMMs). The orders of these messages are crucial to the system, thus these messages have to be delivered under the total ordering constraint. For example, if a group of DMMs subscribe to a number of sensors, a totally-ordered multicast would be required for each DMM to
receive same ordered reports from multiple sensors. Therefore, in the DMM or actuator group, messages from different senders should be applied with the total ordering constraint.

- Last, any decisions are made after receiving some reports from sensors, thus, these messages (decisions) must be delivered under the causal ordering constraint. Since sensor reports can only reach to DMMs (never to actuators), this constraint is assigned to each DMM for receiving other DMMs’ decisions in the DMM group only.

In summary, we only need to assign ordering constraints to the DMM and actuator groups, since sensors never receive messages from others. For those messages originated from a common sender, we need to assign the FIFO constraint to them; for those originated from different senders, we need to assign them with the total ordering constraint; while only for DMMs that receive both sensors’ reports and other DMMs’ decisions, we assign them with the causal ordering constraint.

4.4 Ordering Protocols

We have assigned three types of ordering on the message passing among DMMs, sensors and actuators. This section will give the implementations of these ordering protocols.

4.4.1 Implementing the FIFO Ordering

FIFO requires messages originated from the same component to be processed at destinations in the same order as they were processed at the original component. We assume that each component processes messages one at a time, i.e., messages are processed at components in sequential order, no concurrent processing at each component.

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 destinations in the sending order. Otherwise, each component keeps a message counter dispatching a sequence number to each message sent out. Subsequently, those messages originated from the same component can be processed at the target components in the same order as that they are sent at their original component, simply by respecting their sequence numbers.

4.4.2 Implementing the Causal Ordering Constraint

General implementation for causal ordering is by using the Vector Timestamp protocol [Fidge 1988] [Raynal 1992], which is described by Listing 4.1. Causal ordering only applies to decision messages ($M_{dec}$) on DMM components. For a m-DMM group $D^m = \{D_1, ..., D_m\}$, a vector timestamp $VT_{Di}$ is created and maintained by the $Di$ (DMM) at its local space, where $VT_{Di} = VT[1..m]$. The basic idea of this protocol is to let each message carry a vector timestamp (VT) representing its causality, which is checked for deliverability at remote sites. The following is the full protocol algorithm.

---

**Listing 4.1 -- Vector Timestamp Protocol**

At $Di$, where $1 \leq i \leq m$, $VT_{Di}$ is initialised as $VT_{Di}[1..m] := \{0, ..., 0\}$. $VT_{Di}$ is updated upon the following events:

**Rule 1:** Assigning $VT$.

If a message $m$ is originated from $Di$, its $VT$ is assigned as:

$$VT_{Di}[i] := VT_{Di}[i] + 1$$

When $Di$ sends $m$ to its receivers, the message carries the current value of $VT_{Di}$.

**Rule 2:** Checking the deliverability of a causal ordering operation.

When a remote message $m$ from $Dj$ carrying $VT_{Dj}$ is received at $Di$:

if ($m \in M_{dec}$)
Chapter 4. Communication Services

if \( VT_{Dj}[k] \leq VT_{Di}[k], 1 \leq k \leq m \) and \( k \neq j \) {
    \( m \) is deliverable;
    \( VT_{Di}[j] := VT_{Di}[j] + 1; \)
} else \( m \) is deferred;
else { // \( m \in M_{dec} \)
    \( m \) is deliverable;
    \( VT_{Di}[j] := VT_{Di}[j] + 1; \)
}

The rule 2 in the protocol means that if a message is a decision, its causality is checked, otherwise, its causality is not checked. A non-decision message is a causally free message, which means it has no cause-effect relation with any other messages at a member and can be processed right away without any delay.

4.4.3 Implementing the Total Ordering Constraint

There are generally two algorithms used to implement total ordering. One is to generate a unique sequence number (USN) for each message delivery. Thus, messages can be processed in a unique sequential order group wide. The other one is the token-ring algorithm. The USN algorithm has two approaches as well: centralised approach and distributed approach. We adopt the simple approach -- the centralised sequencer for implementing the total ordering of message delivery.

Centralised sequencer [Kaashoek 1989] is a straightforward technique by allowing one member to be the USN generator (named sequencer). When a message is originated from a component, the component sends a USN request to the sequencer. The sequencer simply keeps a USN counter that increases by 1 each time a USN request is received. The value of the USN counter is then returned to the component. Then the component attaches the USN to the message.
which is propagated to other components later on.

We use a supervisory DMM as the sequencer. To be able to decide a total-ordering operation is deliverable, each member keeps a variable of USN major in its local space to record the maximum USN executed so far. If a total-ordering operation arrived holds the next 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 4.2 -- Assigning the USN Protocol

Rule 1: Acquiring a USN from the sequencer. At each member site:

while (true) {
    receive (application, m); //m is a total-ordering message.
    send (the sequencer, USN-request);
    receive (the sequencer, USN-reply);
    m.USN := USN-reply.USN;
    multicast (m);
}

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 totally-ordered message is executable. At each member site:

int USN major := 0;
while (true) {
    receive (member, m);
    if m.USN == USN major+1
Chapter 4. Communication Services

m is executable;
else m is deferred;
}

4.5 Reliable Multicasting Service

Communication mechanisms are important to any distributed system. Today, most commercial distributed systems are built on point-to-point communication primitives, typically, TCP/IP and UDP/IP. These point-to-point communication primitives may be reliable like TCP/IP, may be unreliable like UDP/IP. The UDP/IP style multicasting primitive is available for destinations in the same local subnet. But generally, there is no reliable multicasting primitive, which can send a message to multiple destinations in one go, available over a wide area network.

4.5.1 Atomic Multicast

The communication channels created using the UDP/IP datagram transmission have to deal with extra problems, such as message loss, duplication and out of order. On the other hand, the TCP/IP channels guarantee messages transmitted in the right ordering, no message duplication and loss. However, sending a message down to multiple channels is still error-prone, as the sending component may fail in the middle of the sending process, as a result, some components will receive the propagation, some will not. Therefore it is crucial to ensure a multicast primitive that either sends messages to all receivers or none of them, i.e. a property known as multicasting atomicity [Birman 1991] [Hadzilacos 1993].

Multicast primitives have the following types:

- **Atomic multicast**: A message transmitted by atomic multicast is either received by all of the processes that are members of the receiving group or else it is received by none of them. We define the membership of groups so
that failed processes cannot remain as members of any groups.

- **Reliable multicast**: This is a message transmission method that makes a best effort to deliver to all members of a group but does not guarantee to do so.

- **Unreliable multicast**: An unreliable multicast just transmits the multicast message once and there is nothing that is guaranteed.

An atomic multicasting protocol has to guarantee all-or-none semantics. It is a protocol that tolerates the sender's failure in the middle of multicasting. The protocol should guarantee two properties: (1) in the absence of sender's failure, a multicast is received by all operational members; (2) in the presence of sender's failure, a multicast is either received by all operational members or by none of them. Note, we use the term operational members instead of members to ensure that the multicasting will proceed despite any receiver's failure during the transmission.

In order to provide reliable multicast communication in a reactive system, the atomic multicasting is used as the basic communication primitive among sensors, DMMs and actuators. For instance, in order that a group of DMM processes interested in a particular application can make correct decisions in the presence of site crashes, an atomic multicast is required to ensure that either all DMM processes in the group receive the report about this application, or none of them receives the report. This will satisfy the completeness requirement. However, note that in a reactive system group, component members are not all connected to each other, i.e., each member is not connected to all other members but part of them. For example, a sensor may be subscribed by part of DMMs rather all the DMMs, thus its report is only received by part of components, not all.

### 4.5.2 Multicasting Atomicity Protocol

The protocol is implemented by letting the sender multicast sequentially, i.e. one multicast after another. The sender ensures that before starting a new multicast, the previous one is confirmed to have reached all operational members.
Chapter 4. Communication Services

We refer to a multicast that may not be received by all operational receivers as an *unstable* multicast, and a *stable* multicast means that the multicast is received by all operational members. Here operational members mean that they are the receivers of the sender sending.

The implementation is based on the assumption that a reliable FIFO communication can be established between any two members. This is achievable by creating a TCP/IP reliable channel between two members. A multicast is sent by the sender transmitting the same message to group members on a one-to-one basis.

The basic idea of achieving multicasting atomicity is to let remaining operational members, which have connections with the crashed member, exchange their latest multicast received from the crashed member. However, the problem is, in the reactive system group, members of such a group are not equally weighted, e.g., DMMs have greater weights than sensors and actuators because they will either receive multicasts from sensors or send multicasts to actuators, whereas sensors or actuators only send or receive multicasts. Hence, to exchange the latest multicast among group members we must identify the DMMs, sensors or actuators' operations.

Let a supervisory DMM be the coordinator. At first, the coordinator must distinguish the three sub-groups (DMM, sensor and actuator) from the original group. This can be done by each group member joining the group and sending its identification to the coordinator so the coordinator can put it into the corresponding sub-group. The atomicity protocol will start its scenario according to the following events:

- **A sensor crash.** If a sensor site crashes, it will affect the DMM sub-group. Also this sensor may not be subscribed by all DMMs, the coordinator has to extract a subgroup again from the DMM group, which is comprised of DMMs subscribing to this sensor. Let \( G_d = \{ d_1, d_2, \ldots, d_n \} \) be the group which members subscribe to the crashed sensor.

- **A DMM crash.** If a DMM site crashes, it only affects the actuator sub-group
because it does not send messages to sensors. This DMM may not subscribe to all actuators, thus similarly the coordinator has to extract an actuator group which members have connections with the crashed DMM. Let \( G_a = \{a_1, a_2, \ldots, a_n\} \) be the group.

- **An actuator crash.** The crash of an actuator site does not affect the group communication since it will not send any messages to any other members. Thus the atomicity protocol will not deal with the actuator crashes.

After the coordinator has identified the group \( G_a \) or \( G_d \) which has connection with the crashed site, the multicast atomicity protocol can be started. Let \( G = \{g_1, \ldots, g_n\} \) be the members of the group either \( G_a \) or \( G_d \). At \( g_i, i = 1, \ldots, n \), it keeps a vector of the latest multicast from every other members: \( mcast[g_1, \ldots, g_n] \), where \( mcast[g_k] \) is the latest multicast message from \( g_k \), \( k = 1, \ldots, n \). Upon the reception of a message \( m_i^{g_k} \) from site \( g_k \) at \( g_i \), \( mcast[g_k] := m_i^{g_k} \).

When a member first detects the failure of \( g_k \), it sends an \( m_{atom}(g_k) \) message to the coordinator. Upon receiving this message, the coordinator initiates the multicasting atomicity protocol by multicasting this message to the group. Each group member that receives messages from \( g_k \) will reply to the coordinator by attaching its latest message received from the failed member after receiving the \( m_{atom}(g_k) \) from the coordinator. The coordinator then collects these messages which each member received from the failed member. Since multicasts are sent in sequential order, the collected messages would be either the \( m_i^{g_k} \) or the \( m_{i,j}^{g_k} \). Based on this information, the coordinator is able to conclude that the \( m_{i,j}^{g_k} \) is the unstable message, and finally multicast it to all of these operational members. Then these operational members receive the latest multicast from the failed member.

Here we give the full multicasting atomicity protocol:

---

**Listing 4.3 -- Atomic Multicasting Protocol**

**Step 1:** The coordinator identifies the DMM, sensor and actuator sub-group.
Chapter 4. Communication Services

Step 2: The first member $g_i$ who detects the crash of $g_k$ sends an $m_{atom}(g_k)$ message to the coordinator.

Step 3: The coordinator gets the target group ($G_o$ or $G_d$) which are related to the crashed site and then multicasts the $m_{atom}(g_k)$ to all group members.

Step 4: Upon receiving the $m_{atom}(g_k)$ from the coordinator, each non-coordinator member $g_i$ that has a communication channel with $g_k$ and receives messages from it, replies an $m_{atom-reply}(g_i; mcast[g_k])$ message to the coordinator, $mcast[g_k]$ is the latest multicast received from $g_k$ and kept at $g_i$.

Step 5: Upon receiving messages of $m_{atom-reply}(g_i; mcast[g_k])$ from all non-coordinator members, the coordinator concludes that the unstable multicast is the $(m_{i+1}^{sk})$ from the crashed member, and multicasts an $m_{atom-comm}(m_{i+1}^{sk})$ message to the group.

Step 6: Upon receiving the $m_{atom-comm}(m_{i+1}^{sk})$ from the coordinator, each non-coordinator member that received messages from $g_k$ adds the $m_{i+1}^{sk}$ to its message buffer.

The advantage of this protocol is that, in the absence of failure, no extra message is needed for preventing the sender failure, except a $n$-vector is used at each member to store the latest multicasts received from other group members.

4.6 Membership Management

Both reliable and ordered multicast services, discussed above, are based on the group membership management. The group membership can be changed by members leaving or joining, especially by crash leaving [Birman 1994]. When the membership changes, the system may suffer from incorrect scenarios. For example, a sensor crash failure leads to a membership change and causes its subscribers receiving incomplete reports and the system generating errors.

Membership is the information about a group that shows who are currently in
the group, in other words, who are operational apparently. The membership is changed by a join/leave event (removing a crashed site from the group is considered as a leave event). In practice, the membership \( G = \{g_1, \ldots, g_n\} \) is replicated at each member so that a multicast can get the membership locally and be able to send messages to those who are currently in the group. Dynamic membership can be based on a predefined set of sites and the membership at a time is a subset of this predefined set of sites. But theoretically, the membership can be based on an infinitive set of sites, provided all sites can be fully connected with each other.

As one of the design goals for a distributed system is fault-tolerance, we have to ensure the correctness of the reactive system model even in the presence of crash failures. Service availability and decision correctness should be maintained within the rest of the system in the presence of crash failures. In the case of a sensor or actuator failure, it is simple to deal with. We can invoke the multicast atomicity protocol and then remove this crash member from the system. In the case of a DMM failure, we can either use a replication technology to replace its work, or remove it from the system and ensure the correctness within the rest of the system by invoking the atomicity protocol as well. In a word, the crashed component should be removed from the system in a way that the system is transferred from one consistent state to another consistent state. Removing a crashed member from the system is achieved through changing the membership of the group. The membership changes can also be triggered by members joining and leaving. Therefore, the group membership management is a mechanism to make the system consistent and fault-tolerant.

We use an object view to represent the membership. The view is initialised to contain a set of initial members \( \{g_1, \ldots, g_n\} \), and is encapsulated with two basic operations, leaveGroup() and joinGroup(), to remove a member from the view and to add a member to the view respectively. View change events are delivered in total order at all members. Members start from the same initial view \(-\ view_0\). And view_i, where \( i = 0, 1, 2, \ldots \), represents the \( i \)-th view of the successive views; view_i and view_{i+1} are different only by an addition or a deletion of one member. A
view update event (leave/join) is multicasted to the members by the coordinator with $view_{i+1}$ replacing $view_i$, i.e. changing $view_i$ to $view_{i+1}$.

Each join/leave event is sent to the coordinator first, from there the event is multicasted to other members. This will ensure the same set of view change operations to be delivered in total order at all members. [Wang 1999] has presented the corresponding protocols for membership changes by a join/leave event including a crash leaving.

4.7 Summary

Communication mechanisms are important to any distributed system. This chapter discussed the group communication services used in the reactive system model. Multicast atomicity and ordered constraints are introduced to guarantee the correctness of the reactive semantics. The correctness of the reactive system model is subject to the crash failures, which lead to the incomplete communication in the reactive components group. To overcome this, the atomic multicast communication is invoked. It guarantees the complete communication, i.e., group members either all receive the message sent from a sender, or none of them receives, and provides the reliable communication. The atomic multicast is necessary in the reactive system model, because each DMM has to get the complete information and then makes correct decisions.

On the other hand, when different messages are sent to the different group members concurrently, their arriving orders may be different. This is due to different network latencies on communication links between the members, and different speeds of machines on which the group members are running. For example, if two sensors multicast to a group of DMMs at about the same time, their reports may not arrive in the same relative order at members of the group due to one report has dropped and been retransmitted. The different orders of messages may have fatal effect on decision making. To ensure the correct decision making, three ordering constraints are applied in the reactive system.
model: FIFO ordering, total ordering and causal ordering. The FIFO ordering is the simplest and default ordering for the most applications, while the total ordering is the strongest ordering. We have categorised three types of message sets in the system and assign them with these three ordering constraints.

Both atomic and ordered multicast services are based on the group membership management. The membership can be changed by members leaving or joining, especially by crash leaving. When the membership changes, the system may suffer from incorrect scenarios. Membership management provides a service to maintain the system consistent and fault-tolerant. According to the leave/join events causing the membership changes, [Wang 1999] has presented the corresponding protocols to maintain the system consistent. The membership protocols are based on the membership management service (GMS) developed by the ISIS project [Birman 1994] [Birman 1996]. The GMS supports the maintenance of group state consistency, despite random communication delays, failures and recoveries. The membership service maintains the mapping from the group address (identifier) to the group membership, and also achieves agreements among group members upon membership changes.
Chapter 5

Implementation Issues

We have presented the formal reactive system model and the communication services between the reactive modules in Chapter 3 and 4 respectively. This chapter will look into the issues involving the implementation of such a system.

One important issue in the implementation of the reactive system model is the communication between DMMs, sensors and actuators. As discussed in the previous chapter, the communication between these entities is much intricate. Many issues have been raised there, e.g., various constraints and requirements have to be ensured; the synchronization must be guaranteed, etc. We will use the Java programming language to implement these features in this thesis, because 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] [Arnold 1996]. With the Java language, DMMs, sensors and actuators can be implemented as generic Java classes distributed all over the networks, therefore the DMMs can know the current states of applications (from the sensors attached to them), and then use the actuators to change their states.
5.1 System Framework

The implementation of a reactive system can be represented by a layered framework, as depicted in Figure 5.1 [Boasson 1998]. It consists of four layers on the basis of the network application interface, reactive control protocol, reactive schemes, and application programs. This layered framework separates the concerns, and promotes modularity and flexibility.

![Layered Framework Diagram]

Figure 5.1: The Layered Framework

- **Application programs.** At this layer, only application level functions are concerned. The programs are user interfaces to the reactive system. The system implements a set of reactivities exported to users. Issues about the reactive control should not be considered.

- **Reactive schemes.** This layer concerns about what reactive schemes should be adopted. The schemes are the questions of using what policies for DMMs and what sensors and actuators. These questions should be answered based on the specific applications.

- **Reactive control protocols.** The layer implements all the protocols for the generic reactive system. The protocols implement all the constraints and coordinate the software entities (DMMs, sensors/actuators) to provide an integrated behavior in order to achieve system availability and fault-tolerance.
Chapter 5. Implementation Issues

- **Network application programming interfaces (API).** The reactive control protocols are implemented based on the network API primitives provided by underlying operating systems. The network API often includes the primitives to both reliable and unreliable point-to-point network communications. Based on what communication primitives being used, this layer will decide what assumptions about communication properties can be assumed, such as message transmission delay, any message loss during the transmission, message ordering, and so on.

In Figure 5.1, application programs and reactive schemes are the responsibilities of application developers since they are application-specific programs. The implementation of the generic reactive system model in this thesis involves the reactive control protocols and the network application programming interfaces.

<table>
<thead>
<tr>
<th>Classes</th>
</tr>
</thead>
<tbody>
<tr>
<td>Mini-Protocol Objects</td>
</tr>
<tr>
<td>Utility Objects</td>
</tr>
<tr>
<td>Java Virtual Machine</td>
</tr>
</tbody>
</table>

Figure 5.2: The Object Layers -- Utility objects are built above Java virtual machine, mini-protocol objects are built above the utility objects, and root classes are built about both utility objects and mini-protocol objects.

From our design in Chapter 3, we know that the main task of implementing the reactive system model is to implement DMM, sensor and actuator classes respectively, and it is very complicated. The complexity stems from the control protocols that have to be embedded in these classes. To ease the construction of these classes, we can divide them into several layers and build layered object
Chapter 5. Implementation Issues

systems to assist constructing them, i.e., a class (DMM, sensor, or actuator) is implemented based on the layered object system [Barcio 1995], as depicted in Figure 5.2. According to these layered objects, each class (DMM, sensor or actuator) can be understood as a composite object of lower layer objects.

One important issue involved in the implementation of a distributed system is the communication mechanism used between distributed objects. There are two types of communication methods provided in Java Virtual Machines [Harold 1997]:

- **Unreliable datagram.** Datagram is a common unreliable communication service which does not guarantee to have any reliability. Messages sent by using such a service may get lost, duplicated and delivered out of order. However most messages will get through. A typical datagram protocol is ARPANET user datagram protocol/Internet protocol (UDP/IP) [Tanenbaum 1989].

- **Reliable stream.** Stream is a common reliable communication service which performs over a virtual circuit channel to provide flow control and reliable, sequenced message delivery. Typical stream protocols include ARPANET transmission control protocol (TCP/IP), and the ISO OSI class 4 transport protocol (TP4) [Tanenbaum 1989]. However, when the channel breaks down, the data in transit may be lost.

In our implementation, the communication between DMM, sensor and actuator classes is implemented with these two patterns: **multicast datagram** and **stream-based communication.** Within each pattern, multicast atomicity and different ordering constraints will be ensured by implementing the corresponding protocols. Using various features of the Java programming language, DMMs, sensors and actuators are implemented as both single and multi-threaded entities. Following sections will discuss the implementation issues of these entities with both multicast datagram and stream-based communication patterns, including various constraints.
5.2 Communication Patterns

5.2.1 Multicast Datagram Communication

The multicast datagram method is a communication mechanism used in the UDP protocol and the group communication. It uses multicast datagrams to implement communication between entities within a single thread entity. Using the single thread of control, a datagram is sent out onto a subnet, where a group of entities are located in, with an address reserved for multicasting, whenever a specific event occurs. Other entities can connect to the subnet simply by creating a multicast socket and join in the group. Figure 5.3 depicts a multicast datagram communication.

![Diagram](image.png)

Figure 5.3: Multicast datagram communication

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, for example, the TTL could be:

- 0: all subscribers on the *same host machine* receive the datagram from the sender.
- 1: all subscribers on the *same subnet* receive the datagram from the sender.
Chapter 5. Implementation Issues

- 32: all subscribers located on any of the subnets to receive packets from the sensors also located on any of the subnets (not necessarily on the same subnet).

However, many routers provide only limited support for multicasting. Packets are allowed to be multicasted within a subnet, but not to pass through the router into other subnets. To overcome this, a tunnelling approach has been invoked, as shown in Figure 5.4. Tunnelling establishes two software entities, one located on a subnet, and another on a remote subnet. On the first 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 class.

Figure 5.4: Tunnelling multicast packets between subnets

5.2.2 Stream-based Communication

The stream-based communication can provide reliable communication between entities. Current TCP/IP protocols used in Internet adopts this communication method. Since each stream-based communication is dedicated to one connection between two entities, a stream-based class must be implemented as a multi-threaded entity to handle multiple connections with a group of entities. Using Java multiple threads, a message is sent out to each receiver using
Chapter 5. Implementation Issues

dedicated connections established between the sender and receivers. Each connection is handled by its own thread of execution. Other entities can subscribe to the sender by simply requesting a connection, which creates a new thread to handle that connection.

In order to synchronize the transmission of reports, each sender creates a ThreadGroup, into which each new thread created to handle a connection is placed. Using the ThreadGroup, the sender can invoke each thread in the group to send to its subscribers at the same time, rather than the threads having to send to the subscribers individually and asynchronously. Figure 5.5 depicts the architecture of a stream-based sender class.

![Diagram showing the generic sender architecture]

Figure 5.5: The generic sender architecture

5.3 Implementing Temporal Constraints

As we defined DMMs, sensors and actuators in Chapter 3, their temporal constraints must be guaranteed, i.e., messages originated from them must arrive at the targets within the certain time periods [Schenke 1994]. However, in a real-time system, due to different network latencies and different speeds of machines, a message multicast to a group from an originator may not arrive at the same time at members of the group. Different members (of the group) may receive the message with different times; some may receive it quickly and some may slowly,
depending on their machines and networks they are running on. Thus, among the messages originated from DMMs, sensors and actuators, some may meet temporal constraints and some may not. Hence, in the implementation of the reactive system, we have to ensure all messages in the group meet the corresponding temporal constraints.

If a message originated from a DMM, a sensor or an actuator does not meet the temporal constraint, that means, it does not arrive the targets within a given time. This is similar to the situation of a member crash leaving and leaves that some targets (group members) may receive the message within the given time, and some may not so as the message arriving after the time constraint is discarded. In this case, we have to apply a protocol, similar to the atomic multicast protocol, to ensure that all the target members of this message are able to receive it finally, as the correctness requirements of the system requires that a message is received either by all its target members or none of them. This situation is not exactly same as the crash failure so the new protocol should be used in the absence of the atomic multicast protocol. We design the following protocol to implement the temporal constraints based on the atomic multicast one.

The basic idea of achieving temporal constraints is to let the target members exchange their latest multicast received from the sender after the given time constraint. First of all, a sender (DMM, sensor or actuator) detects whether all its target members receive the message it sent by requiring an acknowledgment from them within a time constraint (see 3.3, 3.6, 3.8). If there are some members that did not receive the message within the constraint, the protocol starts. Let a supervisory DMM be the coordinator and it identifies the sub-group (DMM, sensor or actuator) related to the sender that does not meet the temporal constraint. The next steps are quite similar to the atomic multicast protocol. We give the full protocol algorithm in List 5.1, where \( m_{\text{atom}}(g_k) \), \( m_{\text{atom-rep}}(g_k) \), \( m_{\text{cast}}(g_k) \), and \( m_{\text{atom-commd}}(m_{++}g_k) \) are referred to List 4.1.

---

**Listing 5.1 — Temporal Constraint Protocol**

97
Chapter 5. Implementation Issues

Step 1: The protocol starts when the sender $g_k$ detects some target members did not receive the message within the time constraint and sends $m_{atom}(g_k)$ with a notice to the coordinator.

Step 2: The coordinator identifies the target group which are related to the sender site and then multicasts the $m_{atom}(g_i)$ to all group members.

Step 3: Upon receiving the $m_{atom}(g_k)$ from the coordinator, each non-coordinator member $g_i$ in the group, replies a $m_{atom-repl}(g_i, mcast[g_k])$ message to the coordinator. The $mcast[g_k]$ is the latest multicast received from $g_k$ and kept at $g_i$.

Step 4: Upon receiving messages of $m_{atom-repl}(g_i, mcast[g_k])$ from all non-coordinator members, the coordinator concludes that the unstable multicast is the $(m_{i+1}^{g_k})$ from the sender, and multicasts a $m_{atom-commit}(m_{i+1}^{g_k})$ message to the group.

Step 5: After receiving the $m_{atom-commit}(m_{i+1}^{g_k})$ from the coordinator, each non-coordinator member that received messages from $g_k$ adds the $m_{i+1}^{g_k}$ to its message buffer and this message becomes the latest multicast in the group.

5.4 Simple and Composite Entities

DMMs, sensors and actuators can be simple or composite. Before implementing them, we identify these concepts first. A simple sensor (actuator) can only be directly attached to one application. Figure 5.6 (a) shows a simple sensor/actuator architecture. Here the simple sensor $S$ is attached to the application API and reports some state changes of API 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 API when necessary. A composite sensor (actuator) can consist of multiple sensors (actuators) from multiple applications. For example, Figure 5.6 (b) shows that the DMM uses a composite Sc to monitor state changes of two applications API and AP2, and uses a composite actuator Ac to change some state of API and AP2.
Chapter 5. Implementation Issues

when necessary. The composite sensor $S_c$ consists of two simple sensors $S_1$ and $S_2$ that monitor the state changes of $A_P1$, and another simple sensor $S_3$ that monitors the state changes of $A_P2$. Similarly, the composite actuator $A_c$ consists of three simple actuators $A_1$, $A_2$, and $A_3$.

A composite sensor (actuator) can be decomposed into multiple independent simple sensors (actuators). For example, the composite sensor $S_c$ in Figure 5.6 (b) is composed of three independent simple sensors $S_1$, $S_2$ and $S_3$. $S_1$ and $S_2$ are attached to the same application $A_P1$, while $S_3$ is attached to application $A_P2$. They all report to the DMM, and they can be seen as independent simple sensors (actuators).

![Diagram](image)

(a) A simple sensor/actuator  
(b) A composite sensor/actuator

Figure 5.6: Sensors and actuators

DMMs can also be simple or composite. Similarly, a composite DMM can be decomposed into multiple independent simple DMMs (like simple sensors/actuators), thus, we only discuss simple DMM, sensor and actuator in the thesis (we just call them DMM, sensor or actuator next).

5.5 Java DMM Agent

The Java DMM agent class will implement the following functions: first, it subscribes to sensors by establishing connections with the sensors and then waits for reports from them; Then, upon receiving reports from sensors, it will process
them and make decisions according to the predefined policy. This is the main feature of the DMM agent. The decisions will then be sent to the related actuators to change the relevant applications' states. The generic DMM class will leave reports processing and decisions making empty that will be implemented by specific DMMs.

As most current operating systems provide reliable communication primitives, such as TCP/IP reliable streams, the FIFO ordering constraint is naturally achieved. However, each DMM may receive messages from different sensors or other DMMs, thus the total ordering constraint must be implemented on these messages sent from different senders. To do this, the generic DMM class will implement a function to check every message with the total ordering constraint after the DMM receives it every time.

Any specific DMM class can inherit from the generic DMM class, either with multicast datagram or stream-based communication. The specific DMM classes in the fault-tolerant computing will implement replication based fault-tolerant policies and their variations. In particular, the following policies can be implemented in fault-tolerant computing [Guerraoui 1997]:

- **Primary-backup replication policies.** These policies use a primary replica to receive and response to client requests. Other replicas form the backup will be in warm stand-by or cold stand-by 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, its services will be replaced by other active replicas.

### 5.5.1 Multicast Datagram DMMs

The generic multicast datagram DMM implements one major thread of control for making decisions and sending to actuators, as well as a number of other threads for receiving reports (datagrams) from multiple sensors. It first builds a
group of multicast sockets and joins them in a group to receive information from multiple sensors, then establishes another multicast socket for sending message to multiple actuators. For each socket connecting to a specific sensor, the DMM creates a thread to manage it. These threads are waiting for reports from the sensors.

Within the generic DMM, it implements an empty run() method, which needs to be implemented for each specific DMM to make decisions. To subscribe to sensors on different subnets, the DMM uses courier and publisher methods which can be referred to Figure 5.4.

5.5.1.1 Utility Objects

Utility objects are at the lowest layer; they are based on the network API or some off-the-shelf libraries. Utility objects manage basic resources such as local component group views, communication channels between components, messages that are transmitted among entities, threads that handle concurrent matters to improve the performance, the system information of sites and port numbers, and so on. Utility objects in the DMM class include the following:

- **Group.** A group of objects representing the local view of an operational component. It basically provides two methods, leaveGroup() and joinGroup(), to remove a member of the group when the member leaves and to add a member when a new member joins respectively. In this communication group, it is simple. The member joining and leaving (voluntary) is implemented by using the joinGroup and leaveGroup primitives provided in Java.

- **Communication handler.** This object sets up the Multicast datagram connections from the DMM to sensor members in the group. The multicasting communication method is referred to Section 5.2. Multicasting primitives are encapsulated in this object.

- **TC.** TC implements the time constraint protocol (Listing 5.1) that guarantees each DMM makes decisions and sends to actuators within the certain time
Chapter 5. Implementation Issues

limit. Also TC is encapsulated with isDeliverable() to check the deliverability of each report.

- VT. VT implements the vector timestamp protocol (Listing 4.1) that generates the VT label for each causal operation request. Also VT is encapsulated with isDeliverable() to check the deliverability of each causal operation request.

- USN and USNAssignor. USNAssignor assigns the USN to each report received from different sensors and each decision sent to different actuators. The USN checks the deliverability of each total update request. These two objects implement the USN protocol described by Listing 4.2.

5.5.1.2 Mini-Protocol Objects

Mini-protocol objects are based on utility objects to implement relatively smaller protocols which are the parts of the whole reactive protocol. These objects are same for all classes (DMM, sensor, or actuator).

In the event of a member crashing, leaving voluntarily, or a new member 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-protocol objects corresponding to three types of membership changing events: StateTransfer, CrashAtomicity and VoluntaryLeave. These protocols can be referred to [Wang 1999].

- StateTransfer. It implements the state transfer protocol. The protocol guarantees a new member joining the group in a consistent way.

- CrashAtomicity. It performs the crash atomicity protocol that guarantees the atomic multicasting property when a member crashes. It also updates each member's local view to remove the crashed component.

- VoluntaryLeave. It performs the voluntary leave protocol and updates each member's local view to remove the voluntarily leaving member.
5.5.1.3 The DMM Class

Based on these utility objects and mini-protocol objects, the generic multicasting DMM is implemented as following:

```
Class MulcastDmm extends Thread {
    protected MulticastSocket receiveSocket, sendSocket;
    protected DatagramPacket sendPacket;
    protected InetAddress receiveAddress = null;
    protected String message;
    protected byte[] buffer;

    public MulcastDmm(int[] receive_ports, int send_port) { ... } //MulcastDmm constructor, establishes a multicast socket to connect to actuators and a group of sockets to connect to sensors, creates a number of Receiver threads.

    public void run() { ... } //Its own run method, undefined, needs to be implemented for each specific DMM.

    protected void send() { ... } //Send method, invokes the send socket to multicast a decision to actuators.

    protected void timeConstraint() { ... } //TimeConstraint method, implements the temporal constraint

    public CommHandler extends Thread { ... } //Communication handler, establishes connections with sensors and actuators.

    public Group extends Thread { ... } //Group class, builds and manages the local group view for this sensor

    public VectorTime extends Thread { ... } //Vector timestamp protocol, implements the causal constraint protocol.

    public USN extends Thread { ... } //USN assigner class, implements the total ordering constraint.

    public MiniProObject extends Thread { ... } //Mini-protocol class, implements membership management protocols
}
```

Any specific DMM class can inherit from the above generic DMM class, and only needs to implement the run() method.
Chapter 5. Implementation Issues

- Multicasting primary-backup replication DMM. This DMM is implemented as a stand alone program, but can easily be embedded into another software entity. It inherits from MulcastDmm class by subscribing to an event sensor to the primary replica and a number of polling sensors attached to other replicas in the backup. The run() method is redefined by checking information received from the primary replica whenever it arrives. If the information indicates the primary replica fails then the DMM will make a decision to select and switch on a replica from the backup that is in warm stand-by state and then send the decision to the actuator attached to this replica using the parent send() method.

- Multicasting active replication DMM. Implemented as a stand alone program, but can be embedded into another software entity. This DMM subscribes to a number of timing sensors attached to each replica. The run() method is redefined by checking information received from all replicas whenever they arrive. If the information indicates one of the replicas fails the DMM will make a decision to select a replica from the others to replace the failed replica, and then send the decision to the actuator attached to this replica using the parent send() method.

5.5.2 Stream-based DMMs

Using the Java multi-threads feature, the stream-based DMM class will be implemented as a multi-threaded entity consisting of multiple objects each of which handles a specific task. It first creates a Connector object to build connections with sensors and actuators it subscribes to. For the connection with the sensors, the DMM has to create a number of ports each of which is used to connect to a specific sensor, while it only needs one port to connect to the actuators since it sends decisions to them. The Connector object will create two groups of threads which manage the connections with the sensors and actuators respectively. Hence each connection is handled by its own thread of execution. These threads wait for reports from the sensors (or send decisions to the actuators)
using dedicated connections.

To synchronize the transmission of decisions, the DMM creates a
ThreadGroup object into which it places the group of threads for the connections
with the actuators, as depicted in Figure 5.5. Using the ThreadGroup, the DMM
can invoke each thread from the group to send a decision to all related actuators
simultaneously, rather than the threads having to send to the actuators individually
and asynchronously. The ThreadGroup object belongs to utility object.

Similarly, the stream-based DMM class consists of utility objects and mini-
protocol objects which are interacting with each other. Most of them are similar to
those in the multicasting DMM class but using the stream-based communication
method. The stream-based DMM class has the following interfaces and functions:

Class StreamDmm extends Thread {
    protected CommHandler commHandler;
    protected ThreadGroup member;
    protected Group group;
    protected int[] receive_ports, send_ports;

    public StreamDmm(int[] port) { ...... } //StreamDmm constructor, creates a
    //connector object and a ThreadGroup, and invokes its own run method.

    public void run() { ...... } //Its own run method, undefined, needs to be
    //implemented for each specific DMM.

    protected void send() { ...... } //Send method, invokes each thread within the
    //ActuatorGroup to send a decision to actuators simultaneously.

    protected void timeConstraint() { ...... } //TimeConstraint method, implements
    //the temporal constraint

    public CommHandler extends Thread { ...... } //Communication handler,
    //establishes connections with sensors and actuators.

    public ThreadGroup extends Thread { ... ... } //Thread group class, manages
    //multi-threads each of which represents a connection between the sensor and a
    //DMM

    public Group extends Thread { ...... } //Group class, builds and manages the
    //local group view for this DMM.

public VectorTime extends Thread { ...... } //Vector timestamp protocol,
  //implements the causal constraint protocol.

public USN extends Thread { ...... } //USN assigner class, implements the total
  //ordering constraint.

public MiniProObject extends Thread { ...... } //Mini-protocol class, implements
  //membership management protocols
}

Stream-based primary-back replication and active replication DMMs are
similar to multicast datagram ones, except they use different communication
channels. They inherit from the StreamDmm class and redefine the run() methods
which are same as ones in multicast datagram DMMs.

5.6 Java Sensor Actor

The generic Java sensor actor class implements the functions of monitoring
events and reporting to DMMs. It can be subscribed by many entities and capable
of reporting to them simultaneously. The sensor class first builds connections with
its subscribers and then monitors for events and reports to its subscribers once
events occur.

Any specific sensor can inherit from the generic sensor. According to the way
that a sensor performs its work, we can implement three types of sensors
categorised as following [Zhou 1998]:

- **Event sensor.** It reports to its subscribers when the monitored event happens.
- **Timing sensor.** It periodically reports to its subscribers on the state of the
  monitored object.
- **Polling sensor.** It periodically checks some states of applications and then
  reports to subscribers.
5.6.1 Multicast Datagram Sensors

Using multicast datagrams, message sending in the group communication can be implemented in a single thread entity. The generic multicast datagram sensor contains only one thread of control to report to multiple DMMs. Its constructor will create a multicast socket with one port to wait for connection requests from DMMs. Once some events occur the sensor will multicast datagrams to a group of DMMs using this socket.

Synchronization for the sensor sending reports to the DMMs is automatically achieved by multicasting datagrams. The sensor only needs to multicast datagrams containing events information to the group, and the datagrams will reach all DMMs at the same time. Within the generic sensor, it implements an empty `run()` method, which needs to be implemented for each specific sensor. To report to DMMs on different subnets, the sensor uses the courier and publisher methods which are referred to Figure 5.4.

Utility objects in the Java sensor class are similar to those in the Java DMM class. They include the Group, Communication handler, and TC objects. Since a sensor class only sends out messages, it does not need to implement any ordering constraint protocols. Mini-protocol objects are same as those in the DMM class. Therefore, the sensor class is constructed by aggregating and initializing the necessary lower layer objects which are instantiated from their classes, and linking the service object. After the initialization, the sensor starts all its child threads, and enters the iteration of handling deliverable messages.

The generic multicasting sensor has following functions:

```java
class MulticastSensor extends Thread {
    protected MulticastSocket sendSocket;
    protected DatagramPacket sendPacket;
    protected InetAddress sendAddress = null;
    protected CommHandler colmhandler;
    protected Group group;
    protected MiniProObject mpo;
    protected String message;
    protected byte[] buffer;
```
Chapter 5. Implementation Issues

```java
public MulcastSensor(int[] receive_ports, int send_port) { ...... }
//MulcastSensor constructor, builds a multicast datagram socket to wait for
//connection requests from DMMs, and establishes the multicasting and local
//host addresses and formats the packet ready for transmission.

public void run() { ...... } //Its own run method, undefined, needs to be
//implemented for each specific sensor.

protected void report() { ...... } //Report method, invokes the socket to multicast
//a datagram to DMMs.

protected void timeConstraint() { ...... } //TimeConstraint method, implements
//the temporal constraint

public CommHandler extends Thread { ...... } //Communication handler,
//establishes connections with all DMMs.

public Group extends Thread { ...... } //Group class, builds and manages the
//local group view for this sensor

public MiniProObject extends Thread { ...... } //Mini-protocol class, implements
//membership management protocols
}
```

Specific sensors can inherit from the generic sensor and only need to fulfil the
`run()` method.

- **Multicast datagram event sensor.** The implementation of this sensor
  requires it to be embedded within an application. It inherits from
  MulcastSensor class. As this sensor only needs to report to DMMs once
  events occur, it does not need to rewrite the `run()` method (which
  automatically runs) in its parent class, rather defining a public `trigger()`
  method to trigger a reporting action once the sensor captures some events. The
  `trigger()` method is used by the application to invoke the sensor to transmit a
  datagram to its subscribers about an event occurred.

- **Multicast datagram 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 MulcastSensor class and rewrites the `run()`
method. This method emulates a timer by putting the thread of execution into sleep for some predefined period of time. Once this interval expires, the sensor transmits its datagram, which is multicast on the local subnet, and then is put back to sleep.

- **Multicast datagram polling sensor.** This sensor is implemented as a stand-alone program, and inherits from the MulticastSensor class. The `run()` method rewritten in the polling sensor also emulates a timer by putting the thread of execution into sleep for some 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.

### 5.6.2 Stream-based Sensors

The generic stream-based sensor class is implemented as a multi-threaded entity consisting of multiple objects each of which handles a specific task. It first creates a Listener object which builds a communication socket with one port and listens to connection requests from its subscribers (DMMs). The Listener object then creates a group of threads to handle the connections with the DMMs through this port. Each thread will manage a specific connection to a DMM, thus each connection is handled by its own thread of execution. Once an event occurs, the sensor will capture it and send to each DMM using dedicated connections. Other DMMs can subscribe to the sensor by simply requesting a connection to it.

In order to synchronize the transmission of reports, the sensor class creates a `ThreadGroup` object, into which all threads from the group are placed. Using the `ThreadGroup`, the sensor can invoke each thread within the group to report to its DMMs at the same time, rather than the threads having to report to the DMMs individually and asynchronously. In addition, the `ThreadGroup` approach places the monitoring of events in one place within the sensor, rather than each thread having to monitor for an event, which duplicates processing. The synchronization architecture is shown in Figure 5.5.
Chapter 5. Implementation Issues

The utility objects of the Java stream-based sensor class include all same objects as those in the multicast datagram sensor, plus a ThreadGroup object to manage multi-threads created in the sensor class. This object is 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.

The generic stream-based sensor class offers the following services:

```java
Class StreamSensor extends Thread {
    protected CommHandler commHandler;
    protected ThreadGroup member;
    protected Group group;
    protected MiniProObject mpo;
    protected int port;

    public StreamSensor(int port) { ...... } //StreamSensor constructor, creates a
    //listener object and a ThreadGroup, and invokes its own run method.

    public void run() { ...... } //Its own run method, undefined, needs to be
    //implemented for each specific sensor.

    protected void report() { ...... } //Report method, invokes each thread within the
    //ThreadGroup to report events to DMMs simultaneously.

    protected void timeConstraint() { ...... } //TimeConstraint method, implements
    //the temporal constraint

    public ThreadGroup extends Thread { ...... } //Thread group class, manages
    //multi-threads each of which represents a connection between the sensor and a
    //DMM

    public CommHandler extends Thread { ...... } //Communication handler,
    //establishes connections with all DMMs.

    public Group extends Thread { ...... } //Group class, builds and manages the
    //local group view for this sensor

    public MiniProObject extends Thread { ...... } //Mini-protocol class, implements
    //membership management protocols
}
```

110
Chapter 5. Implementation Issues

The implementations of stream-based event sensor, timing sensor and polling sensor are similar to multicasting ones, except they use different communication channels. They inherit from the StreamSensor class and redefine the run() methods which are same as ones in multicast datagram sensors.

5.7 Java Actuator Actor

The Java actuator class is simple. It receives decisions from DMMs, and then performs a function to change an application’s state according to the decisions. After that it may return an acknowledgment to the DMMs. In the actuator class, there is no need to deal with synchronization problem.

Each actuator may receive messages from different DMMs, thus the total ordering constraint must be implemented on these messages. To do this, the generic actuator class will implement a function to check every decision message with the total ordering constraint after receiving it every time.

There are two specific types of actuators in the fault-tolerant computing:

- **Warm start-up actuator.** It switches on an object, in particular, a server, currently on warm stand-by.

- **Cold start-up actuator.** It starts up a cold stand-by object and switches it into action to take over the services provided by a current object.

5.7.1 Multicast Datagram Actuators

The generic multicast datagram actuator implements one major thread of control for making actions to change applications, and a number of other threads for receiving decisions (datagrams) from multiple DMMs. Its constructor builds a group of multicast sockets and joins them in a group to receive decision information from multiple DMMs. For each socket connecting to a specific DMM, the actuator creates a thread to manage it. These threads are waiting for decisions from the DMMs.
Chapter 5. Implementation Issues

The generic actuator implements an empty \textit{run()} method, which needs to be implemented for each specific actuator to make actions. The following interfaces are provided in the actuator class:

Class MulticastActuator extends Thread {
    protected MulticastSocket receiveSocket;
    protected DatagramPacket receivePacket;
    protected InetAddress receiveAddress = null;
    protected String message;
    protected byte[] buffer;

    public MulticastDmm(int[] receive_ports) { ...... }  //MulticastActuator constructor,
    //builds a group of multicast sockets to connect to DMMs, and creates a group of
    //Receiver threads.

    public void run() { ...... }  //Its own run method, undefined, needs to be
    //implemented for each specific actuator.

    protected void timeConstraint() { ...... }  //TimeConstraint method, implements
    //the temporal constraint

    public CommHandler extends Thread { ...... }  //Communication handler,
    //establishes connections with sensors and actuators.

    public Group extends Thread { ...... }  //Group class, builds and manages the
    //local group view for this DMM.

    public USN extends Thread { ...... }  //USN assigner class, implements the total
    //ordering constraint.

    public MiniProObject extends Thread { ...... }  //Mini-protocol class, implements
    //membership management protocols
    }

Any specific actuator can inherit from the generic actuator classes and only needs to re-write the \textit{run()} method.

5.7.2 Stream-based Actuators

Similarly, the generic stream-based actuator class is implemented as a multi-threaded entity. It first creates a Communication handler object to build
connections with DMMs with which it has interests. The actuator has to create a number of ports each of which is used to connect to a specific DMM. The CommHandler object will create a group of threads (Receiver) which manage the connections with the DMMs, thus each connection is handled by its own thread of execution. These threads wait for decisions from the DMMs using dedicated connections.

The following codes implement the stream-based actuator class:

```java
Class StreamActuator extends Thread {
    protected CommHandler commHandler;
    protected ThreadGroup member;
    protected Group group;
    protected int[] ports;

    public StreamDmm(int[] port) { ...... } //StreamActuator constructor, creates a
                                          //listener object, and invokes its own run method.

    public void run() { ...... } //Its own run method, undefined, needs to be
                                //implemented for each specific actuator.

    protected void timeConstraint() { ...... } //TimeConstraint method, implements
                                              //the temporal constraint

    public CommHandler extends Thread { ...... } //Communication handler,
                                               //establishes connections with sensors and actuators.

    public ThreadGroup extends Thread { ...... } //Thread group class, manages
                                               //multi-threads each of which represents a connection between the sensor and a
                                               //DMM

    public Group extends Thread { ...... } //Group class, builds and manages the
                                               //local group view for this DMM.

    public USN extends Thread { ...... } //USN assigner class, implements the total
                                           //ordering constraint.

    public MiniProObject extends Thread { ...... } //Mini-protocol class, implements
                                                 //membership management protocols
}
```

113
5.8 Properties Checking

We have implemented DMM, sensor and actuator classes above. As mentioned before, after the implementation we have to check the system assertions (properties) to ensure they are correct or not. These properties are mainly classified as safety and liveness properties, as discussed in Section 3.6.1 (Assertion 3.1 – 3.5). As most of these properties are derived from the temporal constraints, if the temporal constraints are guaranteed, these properties are ensured. In Section 5.3 we have implemented the temporal constraint protocol, which guarantees that DMMs, sensors and actuators all meet the constraints, thus the proof of these assertions is simple.

5.9 Summary

This chapter has presented the implementation of the reactive system model using the Java programming language. First of all, we discussed the framework for the implementation of the system. We found that, the main task of the implementation is to achieve the generic reactive components: DMM, sensor and actuator, which implements various reactive control protocols. The communication between DMM, sensor and actuator has been implemented with two patterns: multicast datagram and stream-based communication. Hence, these entities are implemented with both multicast datagram and stream-based communication methods. Using various features of the Java language, DMMs, sensors and actuators are implemented as both single and multi-threaded entities which can be distributed over the networks.

The DMM, sensor and actuator classes can be implemented based on a layered object system, which consists of root classes, mini-protocol objects, utility objects and network application programming interfaces. The root classes are the highest level and they are the composite of the lower objects. The mini-protocol objects include smaller protocol objects, such as the group membership
management protocols, etc. The utility objects implement the group communication mechanisms, which include the multicast atomicity protocols and ordering constraint protocols. For the sensor class, since it does not receive any messages from others, the sensor does not need to implement any ordering constraints. For the DMM and actuator class, the total ordering constraint has been implemented on both of them.

The temporal constraints on DMMs, sensors and actuators are ensured in the implementation. These temporal constraints prove the safety and liveness properties of the system, which guarantee that the system is fault-tolerant and our reactive system model is correct.

Finally, DMMs, sensors and actuators are implemented as the generic Java classes. Any specific classes can be achieved by inheriting from them. Several specific DMMs, sensors and actuators used in the fault-tolerant computing have been implemented. The generic Java DMM, sensor and actuator classes can form a software package, therefore, they can be used in any distributed and fault-tolerant applications. The evaluation of these classes will be discussed in the next chapter.
Chapter 6

Performance Study

After the implementation, we have conducted a series of tests to evaluate the performance of the Java DMMs, sensors and actuators. The purpose of these tests is to measure the response time of the system to outside inputs, i.e., the time taken to respond the inputs by DMMs, sensors and actuators which are located in a distributed environment. According to the times they take, we can evaluate the effectiveness of the system running in different environments. Theoretically, we can evaluate the system consisting of as many DMMs, sensors and actuators as possible, but, as the tests are conducted in a collection of networked Sun Sparc machines running in Deakin campuses, we can only conduct the tests within a limited number of machines. To overcome this, we also develop a simulation model for the reactive system. With the simulation tool, we can measure the average response time of the system composed of any composition of DMMs, sensors and actuators.

6.1 Test Setting

The evaluation is based on a metric: the average response time (ART) of the system over input events. Suppose we have a system consisting of $m$ DMMs, $n$ sensors and $u$ actuators, which are fully connected with each other (i.e., $n$ sensors
are fully connected with \( m \) DMMs and so do these DMMs with \( u \) actuators). Assume each sensor takes an average time \( ST_i, 1 \leq i \leq n \), to catch an event and report to DMMs; each DMM takes an average time \( DT_j, 1 \leq j \leq m \), to make a decision and send to actuators; and each actuator takes an average time \( AT_k, 1 \leq k \leq u \), to reach an application (to change its state). We have the following definition of the ART:

**Definition 6.1.** The average response time (ART) of a reactive system is defined as:

\[
ART = \frac{1}{n} \sum_{i=1}^{n} ST_i + \frac{1}{m} \sum_{j=1}^{m} DT_j + \frac{1}{u} \sum_{k=1}^{u} AT_k
\] (6.1)

where \( n \) is the number of sensors; \( m \) is the number of DMMs, and:

- \( ST_i = \frac{1}{m} \sum_{j=1}^{m} t_{ij}, \quad t_{ij} \quad (1 \leq i \leq m) \) is the time a sensor takes to report to a specific DMM.

- \( DT_j = \frac{1}{u} \sum_{k=1}^{u} t_{jk}, \quad t_{jk} \quad (1 \leq j \leq u) \) is the time a DMM takes to make a decision and send to a specific actuator.

- \( AT_k = \frac{1}{u} \sum_{k=1}^{u} t_{ik}, \quad t_{ik} \quad (1 \leq k \leq u) \) is the time an actuator takes to change an application.

We evaluate the ART in cases of the system composed of generic DMMs, sensors and actuators. The simplest situation is the system consisting of only one DMM, sensor and actuator respectively, i.e. \( n = m = u = 1 \). The response time we measure for this case starts from the sensor being triggered and then reporting an event to the DMM, and ends at the actuator receiving a decision message from the DMM. The sensor is embedded into a test application that triggers the sensor to report to the DMM when an empty event (a message) has occurred. The DMM makes a response by simply producing a decision message and then sends the message to the actuator after it receives the report from the sensor. The actuator
does nothing but sending an acknowledgment message back to the DMM.

To simplify the test, we embed the sensor and actuator into the same application so that we can measure the communication between the application and the DMM directly instead of measuring the communications between the sensor and the DMM, the DMM and the actuator respectively. By doing so, we can let \( AT_k = \frac{1}{n} \sum_{i=1}^{n} t_i = 0, \) if \( 1 \leq k \leq u, \) since the actuator is located within the application. Hence, the test measures the overhead of the sensor being triggered by the test application and the time taken for reporting the event to the DMM and the DMM producing a message and then sending to the actuator.

Based on this approach we can measure \( ST_i \) and \( DT_j \) in cases of the system consisting of multiple sensors, DMMs and actuators. For simplicity, suppose there are same number of sensors and actuators \( (n = u) \) in the system. We embed each pair of sensor and actuator into one application. So from (6.1) we have:

\[
ART = \frac{1}{n} \sum_{i=1}^{n} ST_i + \frac{1}{m} \sum_{j=1}^{m} DT_j + \frac{1}{u} \sum_{k=1}^{u} AT_k
\]

\[
= \frac{1}{nm} \sum_{i=1}^{n} \sum_{r=1}^{m} t_i + \frac{1}{mo} \sum_{j=1}^{m} \sum_{j=1}^{u} t_j = \frac{1}{mn} \sum_{i=1}^{n} \sum_{j=1}^{m} (t_i + t_j)
\]

(6.2)

where \( t_i \) is the time a sensor takes to report to a DMM and \( t_j \) is the time a DMM takes to send to an actuator. The total time \( (t_i + t_j) \) is the response time for the composition of one sensor, one DMM and one actuator, and it can be measured using the above method. Thus we can calculate the overall \( ART \) according to Formula 6.2. The \( ART \) will mainly be affected by the following factors:

- **Communication methods.** We have implemented the Java DMM, sensor and actuator with two different communication methods: multicast datagram and stream-based communication. Generally speaking, the multicast datagram communication is unreliable but faster than the stream-based communication.

- **Network environments.** The system running in different network environments will have different affects on the \( ART. \) Generally, the system running in a
local host will have a faster response than it running in a distributed environment because it uses less resource in the local host and the communication time between DMMs, sensors and actuators is shorter.

- **The number of sensors/actuators and DMMs.** Intuitively, more sensors/actuators and DMMs in the system will need more resource and add more overheads on the networks it runs, thus the ART of the system will have a higher value with an increasing number of sensors/actuators and DMMs.

- **Ordering constraints.** From Chapter 4, we get the strength levels of ordering constraints. The general rule for the response time is -- the stricter the set of ordering constraints being placed on the corresponding set of messages, the longer response time and lower throughput rate incur.

The performance study in this chapter is based on the ART of the reactive system. According to the factors affecting the ART, we conducted three groups of tests, each in a different distribution environment. The first group of tests is conducted with DMMs and sensors/actuators located on the same host. The second group is conducted with DMMs and sensors/actuators located on the same subnet but different hosts. In tests with more than one DMM and sensor/actuator, each DMM or sensor/actuator is located on a separate host. The last group of tests is conducted with sensors/actuators located on a remote subnet from the DMMs' subnet and each DMM or sensor/actuator is located on a separate host.

Under each of these groups, we also conducted two groups of tests: the first group using the multicast datagrams communication and the second using the stream-based communication. Within each of these groups, we measure the ART of the system consisting of generic sensors, DMMs and actuators with the varying numbers of them. Each group is tested 1000 times with an average time taken from these results.

The tests are performed on a collection of networked (10M Ethernet) Sun SPARC machines running the Sun Solaris 5.6 operating system, across two Deakin campuses (Geelong and Rusden, 100 km apart). Sun's JDK1.2 Java interpreter [Jaworski 1998] is used to run all modules. The most involved Java
packages are Java network and input/output packages [Arnold 1996] [Harold 1997].

6.2 System Performance

In this section we will analyze the effects of the tests with respect to the communication methods, the network environments, and the number of DMMs or sensors/actuators.

6.2.1 Multicasting Communication

Multicasting DMMs and sensors/actuators use datagrams to multicast messages to a group of receivers. Using the group membership primitives, the messages can be sent to receivers almost simultaneously. A multicasting DMM simply creates a multicast socket and subscribes to sensors by using the joingroup primitive (provided in Java). For every event detected, a multicasting sensor multicasts a message to its subscribers (DMMs), and each multicasting DMM receiving this message will send back a decision message (to actuators). The sensor measures the time taken for this process, i.e., \((t_i + t_j)\) in (6.2). The tests showed that each DMM received packets from the sensors simultaneously.

By measuring \((t_i + t_j)\) and calculating the formula (6.2), we can get an average overhead associated with increasing the number of DMMs and sensors/actuators. From the test results we found that the ARTs with the multicasting communication are not affected by increasing the number of sensors, i.e., each sensor performs its function independently, nothing to do with other sensors. This is because each multicasting sensor uses a dedicated communication channel (multicast socket) with its subscribers thus different sensors are not influenced with each other.

Figure 6.1 depicts the ARTs of the multicasting modules in the case of one sensor/actuator, with an increasing number of DMMs. From this figure we can see
that the multicasting ARTs on a single host have a significant increase with an increasing number of DMMs, while the ARTs on a local subnet and remote subnets have shown that they are not affected by the number of DMMs!

![Graph](image.png)

Figure 6.1: The ARTs with the multicasting communication. The multicasting ARTs on a local subnet or remote subnets are not affected by increasing the number of DMMs and sensors/actuators.

The reason for this is that, in the cases of local subnet and remote subnets testing, each multicasting DMM running on a separate host subscribes to sensors and receives reports by simply creating a multicast socket and using the jointgroup primitive to join a group, hence, it will not add an extra overhead on the network; therefore, the number of DMMs will not affect the average response time of the system. While in the case of local host testing, all modules are running on a single machine, thus more DMMs will require more resources from the machine and the response time will take longer.

Figure 6.1 has also shown an expected graduate increase of the ART of the multicasting communication as the sensors/actuators are located further and further away from the DMMs. In particular, we noticed the following observations:
Chapter 6. Performance Study

- First, at the very beginning, i.e., in the case of 1 or 2 DMMs, the ART on a local host is the lowest. However, since it increases with an increasing number of DMMs, the ART on a local host overtakes that on a local subnet or remote subnets respectively after the number of DMMs increases to 3 or 9.

- Second, the ARTs on both a local subnet and remote subnets are not affected by increasing the number of DMMs. They stay the same value no matter what number of DMMs is. However, they have a big difference, i.e., the ART on remote subnets is much greater than it on a local subnet. This is because there is an added overhead of tunneling datagrams occurred from one subnet to another, in the case of remote subnets testing.

6.2.2 Stream-based Communication

Stream-based DMMs and sensors/actuators are multi-threaded entities. They use a separate thread to manage each specific connection between a DMM and a sensor/actuator. Each sensor invokes its threads to report to its subscribers (DMMs) when an event is detected. Each thread measures the time \((t_i + t_f)\) which the sensor needs to report to and receive from each DMM. The tests showed that each stream-based DMM received reports from the sensors one after another, rather than simultaneously as a subscriber to the multicasting sensors did.

Table 6.1 shows the ART needed by different compositions of DMMs and sensors/actuators tested on a local host. From this table we found:

- **Rows.** These are test results with multiple sensors reporting to a certain number of DMMs. To our surprise, the ARTs of the system did not increase with an increasing number of sensors. The values on each row are very similar. That means, the ART is not affected by the number of sensors. This is because each stream-based sensor uses a dedicated socket to connect with its subscribers (DMMs) and different sensors use different sockets so that their communications with their subscribers are not influenced by each other.

- **Columns.** These are test results with multiple DMMs subscribing to a fixed
number of sensors. The results confirm our intuition that as the number of
subscribers increases, the ART of the system increases as well. This is due to
that a sensor subscribed by more subscribers is relatively heavy and requires
more system resources such as more threads of executions.

Table 6.1: The ART (ms) tested on a local host. \( M \) – the number of DMMs; \( N \) – the
number of sensors/actuators.

<table>
<thead>
<tr>
<th>( M ) ( \backslash ) ( N )</th>
<th>1</th>
<th>2</th>
<th>3</th>
<th>5</th>
<th>10</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>32.310</td>
<td>30.796</td>
<td>33.088</td>
<td>32.952</td>
<td>33.442</td>
</tr>
<tr>
<td>5</td>
<td>51.318</td>
<td>51.456</td>
<td>51.630</td>
<td>51.942</td>
<td>50.704</td>
</tr>
<tr>
<td>10</td>
<td>105.834</td>
<td>105.706</td>
<td>106.422</td>
<td>106.656</td>
<td>105.798</td>
</tr>
</tbody>
</table>

Therefore, the tests on a local host show that the ART of the system increases
with an increasing number of DMMs, but not with sensors’. We may use one of
the columns to represent the ART of a local host testing.

Other two groups of tests are conducted with DMMs and sensors/actuators
located on different hosts but on the same subnet and remote subnets,
respectively. These tests show a similar result to the local host testing, that is,
with an increasing number of DMMs, the ART increases as well, but not with an
increasing number of sensors. Figure 6.2 shows the comparison of three groups of
tests in the cases of one sensor.

From Figure 6.2, we have three observations.

- First, we see that the time curve for a local host is steeper than those for a
  local subnet and remote subnets. That means, in the tests on a local subnet or
  remote subnets, the ART does not increase very dramatically with an
  increasing number of DMMs. In fact, the ART on both cases increases much
  less than it does on a local host testing! The reason for this is that, each DMM
or sensor/actuator runs on an individual host in the case of local subnet or remote subnets testing, while for local host testing, all DMMs and sensors/actuators are executed on one host, thus with an increasing number of DMMs, the overheads on the local host increases significantly whereas there is only a small network overhead occurred on a local subnet or remote subnets. This is a good result for the reactive system model, because most reactive systems are distributed systems and their DMMs require reports from sensors located in different hosts. This result shows that reactive components running in a distributed environment is more effective than them running in a local host.

![Graph](image)

Figure 6.2: The ARTs on three groups of tests

- Second, we find that the DMMs and sensors/actuators running on remote subnets take more time than them running on a local subnet. But on both cases the ART they took increases at an almost same speed with an increasing number of DMMs. This is because, the communication among remote subnets has more overheads and requires more system resources than that on a local subnet.

- Last, we observe that, in the case of 1 DMM, sensors running on a local
subnet or on remote subnets use more time to report to and receive from the DMM than they use running on a local host. When the number of DMMs increases to 2, there is little difference for the time taken among the three groups. However, when the number of DMMs increases further, the DMMs and sensors/actuators running on a local subnet or on remote subnets use less time than they use running on a local host! This can be explained by the same reason in the first point.

6.2.3 Discussion

The above test results show that the ART of both multicasting and stream-based communications is not affected by the number of sensors. Therefore, we can compare the effects of both communication methods with respect to different distributed environments, as depicted in Figure 6.3 and Figure 6.4.

![Graph comparing ART (ms) with different numbers of DMMs]

Figure 6.3: The multicasting and stream-based testing on a local host

From these figures, we observed the following points:

- First, on a local host, the multicasting modules take less time for response than the stream-based modules. Particularly, the increasing speed of the ART
of the stream-based system is much quicker than that of the multicasting system. The reason is that the multicasting DMMs and sensors/actuators are relatively light, not requiring multiple threads of executions which need additional system resources, whereas the stream-based modules do.

Figure 6.4: Multicasting and stream-based testing. (L - Local subnet; R - Remote subnets).

- Second, on a local subnet and remote subnets testing, the ARTs of the multicasting in both cases stay at the same value while the ARTs of the stream-based have a graduate increase, with increasing the number of DMMs. From Figure 6.4, we can see that the ART of the multicasting on a local subnet is the smallest, whereas that on remote subnets is the greatest before the number of DMMs goes to 7. After DMMs increase to 7 or up, the ARTs of the stream-based on both local subnet and remote subnets testing are greater than that of the multicasting on remote subnets. This can be explained by the same reason in the first observation as well.

- Last, from a local subnet testing to remote subnets testing, there is a big jump of the ART for the multicasting communication, whereas a little increases for
the stream-based communication. This is because there is an added overhead of tunneling datagrams occurred from one subnet to another, for the multicasting communication while the stream-based communication has no such an overhead.

From the above observations, we conclude that multicast datagram is a common unreliable communication service which does not guarantee to have any reliability. Messages sent by using such a service may get lost, duplicated and delivered out of order. Therefore, multicast datagrams may have limited applications in distributed systems covering multiple subnets. However, from above discussions we found that within a local subnet, the multicasting communication is the fastest way to respond events and its ART is not affected by the number of DMMs or sensors/actuators.

Stream-based communication is more reliable and may be more suited to distributed systems than multicast datagrams. The stream-based DMMs and sensors/actuators running in a distributed environment are more effective than they running in a centralized system. They do not have tunneling overhead when covering multiple subnets. Thus we recommend stream-based communication as the implementation method for a distributed computing system covering multiple subnets.

### 6.3 Effects of Ordering Constraints

We have discussed the effects of the communication methods, the network environments, and the number of DMMs or sensors/actuators affecting to the ART above. Besides these factors, ordering constraint is another one to affect to the ART. In this section we discuss the impacts of different ordering constraints in the reactive system model.

Since the FIFO constraint in the implementation of the reactive system is automatically achieved, the main ordering constraints in the system are the totally ordering constraints applied in the DMM sub-group and the actuator sub-group.
The general rule for ordering constraints affecting the response time is -- the stricter the set of ordering constraints being placed on the corresponding set of messages, the longer response time and lower throughput rate incur. Figure 6.5 depicts the ART with the total ordering constraint on DMMs and actuators in the case of the stream-based communication.

Figure 6.5: The effects of the ordering constraints

Figure 6.5 shows a slight increase on ARTs when the ordering constraints are applied. Comparing Figure 6.5 with Figure 6.2, we can find that all the values (ART) on the corresponding values of x-axis are increased. This is because the stricter the set of ordering constraints being placed on the corresponding set of messages, the longer the response time is. We also found that the three curves in Figure 6.5 are steeper than those in Figure 6.2. That means, the ARTs with the ordering constraints increase quicker than those with no such constraints do. This is because more DMMs with the constraints will require more resources.

The tests with the multicast datagram communication show a similar result to Figure 6.5.
6.4 Examples

Up to now, we have demonstrated the performance of the system with different configurations. The testing system is generic, thus the ART of the system mainly consists of communication times between DMMs, sensors and actuators, because they are generic and do nothing on their own. In this section, we try to use specific DMMs, sensors and actuators as examples to demonstrate the system performance.

We introduced and implemented a few specific sensors, DMMs, and actuators in Section 5.4, 5.5 and 5.6 respectively. The specific DMMs will implement primary-backup replication policies and active replication policies, and they are more application-specific. So are the specific actuators which implement warm start-up and cold start-up mechanisms. While the specific sensors implement event monitoring, timing and polling reporting mechanisms, and they are relatively independent to specific applications. Hence we use the specific sensors (event sensor, timing sensor, and polling sensor) as the examples together with the generic DMMs and actuators to conduct the tests. The tests are conducted with the steam-based communication since it is a reliable communication method.

Table 6.2: The ARTs (ms) of three types of sensors (single subscriber)

<table>
<thead>
<tr>
<th></th>
<th>Local Host</th>
<th>Local Subnet</th>
<th>Remote Subnet</th>
</tr>
</thead>
<tbody>
<tr>
<td>Timing</td>
<td>11.526</td>
<td>15.792</td>
<td>18.306</td>
</tr>
<tr>
<td>Event</td>
<td>13.712</td>
<td>16.062</td>
<td>19.608</td>
</tr>
<tr>
<td>Polling</td>
<td>149.721</td>
<td>151.461</td>
<td>164.878</td>
</tr>
</tbody>
</table>

Table 6.2 shows the ART needed by three types of sensors to report to a single DMM. From this table, we found that the polling sensor is slower to report to subscribers. In fact, throughout all the tests conducted, the polling sensor is the slowest one to report to DMMs 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

129
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 effects of multiple DMMs on the reporting times when stream-based timing and event sensors are used to report events to 1 to 10 DMMs. Figure 6.6 shows the results of local host testing (where all timing and event sensors and DMMs are located on the same host) and local subnet testing (where all timing and event sensors and DMMs are located on the same subnet).

Figure 6.6: Event sensor and Timing sensor tests on a local host and local subnet. EventLH -- event sensors running on a local host; TimingLH -- timing sensors running on a local host; EventLS -- event sensors running on a local subnet; TimingLS -- timing sensors running on a local subnet.

Figure 6.7 shows the result of local host testing and remote subnet testing (where all timing and event sensors and DMMs are located on two interconnected subnets (about 100km apart)). These two figures confirm our observations above, i.e., as the number of DMMs increases, the ART of the system increases as well, however, such an increase of time is not very dramatic in case of a local subnet and remote subnets. Besides that, we also have the following two observations:
Chapter 6. Performance Study

- The first observation from these two figures is that the stream-based event sensors take slightly more time to report to DMMs 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 similar to the last point observed from Figure 6.2. That is, at the very beginning, the system running on a local subnet or remote subnets uses more time than it running on a local host. However, when the number of DMMs increases further, the system running on a local subnet or remote subnets uses less time than it running on a local host!

Figure 6.7: Event sensor and Timing sensor tests on a local host and remote subnets. EventRS -- event sensors running on remote subnets; TimingRS -- timing sensors running on remote subnets (EventLH and TimingLH are same as above).

6.5 Simulation Result

We have also designed a simulation to evaluate the reactive system model in the case that it consists of a great number of DMMs, sensors and actuators. The simulation language is Simjava [Howell 1998], which is a discrete event
simulation package written in Java and conceptually based on the Simlibrary for C++ [Howell 1996] [McNab 1996]. The simjava package has been designed for simulating fairly static networks of active entities that communicate by sending and receiving passive event objects via ports. This model is appropriate for hardware and distributed software systems modelling. Since a reactive system is often a distributed system and its components are active entities distributed over networks, this simulation model is quite suitable for simulating our reactive system architecture.

### 6.5.1 Simulation Model

A simjava simulation contains a collection of entities, each of which runs in its own thread. These entities are connected together by ports and can communicate with each other by sending and receiving event objects through these ports. An entity’s behaviour is encoded in Java using its `body()` method. A static central system class controls all the threads, advances the simulation time, and maintains the event queues. The progress of the simulation is recorded through trace messages produced by the entities, and saved in a file.

To simulate the reactive system architecture, we create three types of entities: DMM, sensor and actuator. 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 and actuators as well. Once a DMM makes a decision, the decision will be sent to the related actuators that change the states of applications. In the simulation, sensors send out the events first. The events are sent to each DMM, which will make a decision and send the decision event to all actuators after it receives the events from all sensors once. Each actuator performs an action after it receives a decision event and then feedbacks an acknowledgment event to the DMM from which it receives the decision event.

The entities in the system can be described as follows:

\[
DMM\text{[Constructor(), body(), ports<\rightarrow>]}
\]  

(6.3)
Chapter 6. Performance Study

\begin{align*}
Sensor{[\text{Constructor}(), \text{body}(), \text{ports}<n>]} & \quad (6.4) \\
Actuator{[\text{Constructor}(), \text{body}(), \text{ports}<n>]} & \quad (6.5)
\end{align*}

where \text{Constructor}() is the constructor of each entity; \text{body}() is the behavioral function of each entity; ports<n> means the entity has \(n\) ports.

6.5.2 Execution Time

We have measured the execution time for the communication of DMMs, sensors and actuators to evaluate the performance of the reactive system model. The measuring method is same as that we used in the previous tests. Table 6.3 shows the centralised simulation results for different compositions of DMM, sensor and actuator entities. Here, the "centralised simulation" means that all simulation entities are running in a local host. The results show that as the numbers of DMMs and sensors/actuators increase, the execution times they used to communicate increase as well.

Table 6.3: The execution times (s) for the simulation. \(M\): Number of DMMs; \(N\): Number of sensors (actuators)

<table>
<thead>
<tr>
<th>(M : N)</th>
<th>1</th>
<th>5</th>
<th>10</th>
<th>50</th>
<th>100</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>0.220</td>
<td>0.567</td>
<td>0.593</td>
<td>1.384</td>
<td>2.543</td>
</tr>
<tr>
<td>5</td>
<td>0.525</td>
<td>0.750</td>
<td>0.936</td>
<td>3.658</td>
<td>11.780</td>
</tr>
<tr>
<td>10</td>
<td>0.453</td>
<td>1.010</td>
<td>1.398</td>
<td>9.198</td>
<td>25.638</td>
</tr>
<tr>
<td>50</td>
<td>1.072</td>
<td>3.034</td>
<td>8.929</td>
<td>102.461</td>
<td>636.346</td>
</tr>
<tr>
<td>100</td>
<td>2.268</td>
<td>9.275</td>
<td>25.008</td>
<td>511.678</td>
<td>3136.578</td>
</tr>
</tbody>
</table>

From Table 6.3 we can find that, the values on each row increase more quick than those on each column. That is, the simulation times associated with an increasing number of sensors/actuators increase more rapid than that with an increasing number of DMMs. This is because, increasing the number of sensors/actuators (i.e., increasing applications) will lead to that the simulation has to take more time to handle the actions involved with both the sensors and
actuators, while increasing the number of DMMs only involve DMMs' actions.

Figure 6.8 depicts the simulation results in time curves, where $s$ stands for the number of sensors. We can see 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 an increasing number of DMMs, the simulation time with bigger number of sensors increases more rapid than that with smaller number of sensors.

![Execution time graph]

**Figure 6.8:** The execution times for different compositions

Most reactive systems are distributed systems. To evaluate the reactive system model in a distributed environment, we have to build a distributed simulation, which distributes the system entities on different computers. We can use the extension of the Simjava discrete event simulation package to utilise the Remote Method Invocation (RMI) capabilities of the JDK1.2 to build the distributed simulation [Griffin 1997] [Page 1997].

The distributed simulation shows a similar result to the local host simulation. Figure 6.9 shows the comparison of two groups of simulations. From this figure, we see that the time curves for the local simulation is 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 result
confirms our conclusion obtained from the previous tests that DMMs and sensors running in a distributed environment is more effective than their running on a local host. This is because each distributed entity is running on an individual host for the distributed simulation, whereas for the local simulation, all entities are executed on the same host.

![Graph showing execution time vs. number of DMMs](image)

**Figure 6.9: Two groups of simulations**

From the figure we can see that, in the case of 1 DMM, the distributed simulation uses more time than the local simulation does. When the number of DMMs increases further, their difference is getting little and little. When the number of DMMs increases to 5 or up, the distributed simulation uses less time than the local simulation does!

### 6.6 Summary

In this chapter we presented the performance study of the reactive system model designed in Chapter 3. The evaluation of its performance is based on a series of tests conducted in a collection of networked Sun Sparc machines running in Deakin campuses. The purpose of these tests is to determine the response time
Chapter 6. Performance Study

of the system to outside inputs, i.e., the time taken to respond the inputs by DMMs, sensors and actuators which are located in a distributed environment. According to the times they took, we evaluate the effectiveness of the system running in different environments.

The performance study is based on the average response time (ART) of the reactive system. The main factors affecting the ART are communication methods, network environments, the number of sensors or DMMs in the system, and the ordering constraints on the message passing in the system. According to these factors, we conducted three groups of tests, each in a different network environment. Also, under each of these groups, we conducted two groups of tests each of which is using a different communication method. The test results have shown the following conclusions:

- The ARTs with the multicasting communication are not affected by increasing the number of sensors, i.e., each sensor performs its function independently, nothing to do with other sensors. However, the ARTs with multicast datagram on a single host have a significant increase with an increasing number of DMMs, while the ARTs on a local subnet and remote subnets have shown that they are not affected by the number of DMMs either!

- The ARTs with the stream-based communication are not affected by increasing the number of sensors, but are affected by an increasing number of DMMs. The results show that the ARTs on a local host testing increase quicker than them on a local subnet or remote subnets testing, with an increasing number of DMMs, that means, reactive components running in a distributed environment is more effective than them running in a local host.

- Multicast datagrams are a common unreliable communication service which does not guarantee to have any reliability, therefore, they may have limited applications in distributed systems covering multiple subnets. However, the tests show that within a local subnet, the multicasting communication is the fastest way to respond events and its ART is not affected by the number of DMMs or sensors/actuators. The stream-based communication is more
Chapter 6. Performance Study

reliable and may be more suited to distributed systems than multicast datagrams. It does not have tunneling overhead when covering multiple subnets.

- When ordering constraints are applied, the ARTs with both multicast datagrams and stream-based communication increase slightly. This is because the stricter the set of ordering constraints being placed on the corresponding set of messages, the longer the response time takes.

We also conducted the experiments on specific sensors to evaluate the system performance. They have shown the similar results to the above. To evaluate the system in case it consists of a great number of DMMs, sensors and actuators, we designed a simulation for the system evaluation. The simulation is written in Simjava which is a toolkit for building working models of complex systems. The Simjava simulation contains a number of entities each of which runs in parallel in its own thread. These entities are much similar to the agents and actors we used to model reactive components. The simulation results have confirmed our conclusions obtained from the above tests.
Chapter 7

Applications

The main aim of this thesis is to develop the reactive system model for building better fault-tolerant distributed applications. The stress of this reactive model is to provide a flexible system architecture, which separates fault-tolerant computing polices from fault-tolerant computing mechanisms. In the previous chapters, we have presented the modeling, implementation and evaluation of this architecture. In this chapter we want to apply it to several applications, for the purpose of demonstrating potential benefits of the reactive system model.

Distributed applications range from the provision of general-purpose computing facilities to automated banking and multimedia communication systems [Daniel 1993]. The reactive system model developed in this thesis is very generic, hence it can be used for general-purpose fault-tolerant computing systems, and can be applied to almost any specific distributed applications. The applications discussed in this chapter are: fault-tolerance in a replicated database system, the network partitioning problem solving, and the Web-based teamwork support system. We can see that these applications have demonstrated the great advantage and potential benefits of our reactive system design.
Chapter 7. Applications

7.1 Fault Tolerance in a Replicated Database System

Replication is the maintenance of on-line copies of data and other resources. It is a key to the effectiveness of distributed systems, in that it can provide enhanced performance, high availability and fault-tolerance [Helal 1996]. Replication technology is being used widely and makes fault-tolerance possible. For example, the USENET system maintains replicas of items posted to electronic bulletin boards across the Internet, and replicas are held within or close to the various organizations that provide access to it. Our first application example is a replicated database system.

7.1.1 Problems

In a distributed computing environment, two types of failures may occur in a system [Cristian 1991]: the processor at a given site may fail (referred to as a site failure), and communication between two sites may fail (referred to as a link failure). Site failures may cause the system inconsistent and they are the major threat to the availabilities of the system services. Link failures may result in network partitioning, which is a major threat to the reliability of distributed systems. We discuss fault-tolerance in a replicated database system in the presence of site failures in this example (the network partitioning problem will be addressed in the next application).

In this application, we discuss a replicated database system consisting of two Mini SQL (mSQL) database servers (called replicas) running on two workstations located about 100km apart. In the system 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 7.1 shows the architecture of such a replicated database system [Blaustein 1983], where RP stands for a replication
manager; DB represents a Mini SQL database server.

![Diagram of a distributed replication system](image)

Figure 7.1: A distributed replication system

In this system, each host (Computer 1 or 2) has a replication manager and an mSQL database server running on it. 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 served by the local replication manager 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, site failures may occur in this system. They may cause the system inconsistent and provide incorrect services. As mentioned previously, site failures include crash failures. A crash failure means a process or a component fails at a site. In this example, some decisions have to be made in case of system component failures. Here are some of situations:

- **Case 1: A database server fails.** For example, assume DB1 on Computer 1 fails. In this 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 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 perform it on DB1 when it recovers.

140
Case 2: A replication manager fails. For example, assume RP1 on Computer 1 fails. In that case, all requests have to be submitted through RP2 on Computer 2.

Case 3: A computer fails. For example, assume that Computer 1 fails. In this 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).

In the case 1 and 3, it is essential for a replication manager to know if a database server is alive. It is also essential for a client to know if a replication manager is alive in the case 2 and 3. To achieve these goals, we can embed the reactive system architecture into this application.

7.1.2 Resolutions

We use the Java reactive system modules designed previously to deal with these failures. To do so, we run a Java DMM, a Java sensor and a Java actuator on Computer 1 and 2 respectively. According to the above failure scenario, we describe the resolutions as follows.

- In Case 1, we run two polling sensors on Computer 1 and 2 respectively to report the liveness of database server DB1 and DB2. Two DMMs with primary-backup replication policies are embedded in RP1 and RP2 respectively, and both of them subscribe to both polling sensors and are informed of the liveness of the two database servers periodically, as depicted in Figure 7.2. Once a database server fails, both DMMs will know it, and then make certain decisions (as described before) and use the actuators to instruct RP1 and RP2 to process clients' requests.

- In Case 2, to detect a replication manager failure, we use an event sensor embedded into RP1 to report to the DMMs (same as above) about the failure of the connection between RP1 and DB1. Similarly, another event sensor is
embedded into RP2 to report the failure of the connection between RP2 and DB2. Figure 7.3 depicts this method. If a replication manager fails, that means the connection between this replication manager and its database server fails, the sensor embedded into it will report to both DMMs. Hence they decide all requests have to be submitted through the live replication manager (RP1 or RP2) and use the actuator to operate DB1 and DB2.

![Diagram](image)

Figure 7.2: Using polling sensors – DMM represents a DMM with primary-backup replication policies; PS stands for a polling sensor; A is an actuator.

![Diagram](image)

Figure 7.3: Using event sensors – ES stands for an event sensor.
In Case 3, we use polling sensors (or event sensors) running on each computer to detect computer failures, and run two DMMs with active replication policies on each computer to decide what strategies are used to deal with these failures, as depicted in Figure 7.4. If one of the DMMs cannot receive any message from the polling sensor on another computer within a predefined time limit, it may decide that another computer fails. Thus it will choose another active replica (another computer in this case) to process clients' requests, as described in the previous Case 3.

Figure 7.4: Using the DMMs with active replication policies – DMMa stands for a DMM with active replication policies.

### 7.1.3 Conclusion

Fault-tolerance is an important feature of distributed systems. In this example we use the reactive system approach with the replication technology to provide the fault-tolerance in a distributed database system. In the system, we separate the DMMs and the sensors/actuators. The DMMs stay the same no matter what sensors and actuators are or what changes they have, and vice versa (in Figure 7.2 and 7.3, the DMMs are same whereas sensors are different; in Figure 7.2 and 7.4, the sensors are same while the DMMs are different). This provides a very flexible system architecture in which fault-tolerance can be obtained due to its continuing
work even in the presence of some component failure (e.g., a DMM, or a sensor/actuator failure).

7.2 Network Partitioning Problem

Network partitioning failure is a major threat to the reliability of distributed database systems and to the availability of replicated data [Davidson 1985] [Melliar 1998]. A 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 [Davidson 1984].

A number of diverse solutions have been proposed to solve the network partitioning problem. These solutions can be classified as two kinds of strategies: pessimistic and optimistic [Paeker 1983] [Davidson 1984]. 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 paid enough attention. Therefore, problems may occur when the detection of partitioning cannot be achieved.

Hence it is desirable to implement some mechanisms for both detecting and resolving the partitioning problem. Following sections will address how to use the reactive system approach to detect, analyze and resolve the network partitioning problem. The approach is a relatively centralized method using the Primary/Non-Primary replication control protocol [Paris 1992].

7.2.1 Partitioning Problem

Network partitioning most likely happens at a wide area network. We discuss
Chapter 7. Applications

the same system as in the previous example: a distributed replication system. Assume that the network environment is composed 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 7.5 depicts the architecture of such a distributed replication system.

![Figure 7.5: Structure of a replication system](image)

In this system, each host has a replication manager (called RPM) and a database server (DB) running on it. A client issues transactions through the local replication manager. A transaction request can consist of different subtransactions each of which is to be serviced by a group of servers or replicas. Replication managers which receive the requests from clients divide the transactions into sub-transactions and pass them onto the different replicas. Among one group of replicas, a Primary Replica leads other Non-primary Replicas [Paris 1992]. 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 Replicas will be finalized by either being upgraded to a fully commit mode or downgraded to an abort if conflict exists [Zhong 1998]. Coordination between replica groups is carried out by replication managers to finalize
transactions after collecting results from different service executions.

7.2.2 Partition Tolerant Architecture

Network partitioning happens when gateways between subnets fail. This leads to a situation where server group members distributed in different subnets cannot 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.

For simplicity, we include two subnets connected by one gateway in our network configuration. Figure 7.6 shows the system modeled with the reactive system architecture. In this architecture, each server group member embeds with a sensor that reports its state to DMMs in different subnets. DMMs will decide whether a partitioning happens according to the reports received from the sensors and then make decisions to instruct RPMs how to process transactions using actuators. In Figure 7.6, RPMs function for transaction processing while DMMs function for failure handling and coordination between replica groups.

![The partition-tolerant system architecture](image)

Figure 7.6: The partition-tolerant system architecture
7.2.2.1 Partition Detecting and Notifying

With this architecture, network partition detecting and notifying can be easily achieved. 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.

7.2.2.2 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 finalizing 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
Chapter 7. Applications

type, whether initialized by the P or NP site, and then 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 process 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 cannot 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.

7.2.3 Implementation

The implementation of the DMMs, sensors and actuators used in a replicated system can be achieved by inheriting from the generic classes developed previously and implementing the specific \textit{run()} methods.

7.2.3.1 DMMs

A DMM class will be created and run in a dedicated host in each subnet. Firstly, 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. Secondly, 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 \textit{partition\_detecting()} method to achieve the partition detection:

```java
partition\_detecting() {
    while (true) {
        report\_receiving(); // receive reports from sensors.
        report\_checking(); // check if there are sensors not to report then store them in
        // different vectors for each subnet if there are. If the size of a vector equals to the
        // number of server members in that subnet, then find the DMM address in that
        // subnet.
        confirming(); // sets above DMM as the message sending target and sends out
    }
}
```
Chapter 7. Applications

// checking message and waits for reply. If the DMM replies on time then sets the
// vector for that subnet as zero, otherwise, confirms the partition really occurs
// and invokes partition_notifying() procedure. }
partition_notifying() {
  // broadcast the partition to all server members 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 program for resolving conflict such
as a backout strategy. For the transactions from NP sites for checking and
finalizing, 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 finalize the transaction. This is the primary
first policy which guarantees the primary site interest.

We use transaction_exchanging() and transaction_execution() functions to
achieve the transaction exchanging between DMMs and further processing:

transaction_exchanging() {
  sending(); // send the transaction list to the other DMM, then wait for incoming
  // messages from other DMMs.
  receiving(); // if an incoming message is the confirmation of receiving transactions from
  // the other DMM then delete the transaction object list and break; if an incoming
  // message includes the transaction lists initialised by other P/NP sites then store
  // the transactions and send confirmation to the sending DMM. }

transaction_execution() {
  sending(); // use actuators to send the transaction lists to RPMs to execute
  // transactions.
  wait(); // wait for the message returned from actuators. if the message means a conflict,
  // and the transaction is from P site, decide a strategy to resolve the conflict. if the
  // transaction is from NP site, abort this transaction. if the message means no
  // conflict, inform the NP site to finalize the transaction. }

The detailed implementation of the DMMs can be referred to [Chen 2000(b)].

7.2.3.2 Sensors/Actuators

A sensor object should be created and embedded in each server member.
Sensor objects run and listen to the replicas' states all the time and report to the
Chapter 7. Applications

DMM periodically. Sensors used in the partitioning tolerant system are timing sensors, that is, they periodically report to their subscribers on the states of the monitored objects. Its implementation can be referred to Chapter 5.

An actuator object will be created and embedded in each server member as well. Actuator objects run and wait for the decision message from the DMMs and then instruct RPMs to execute transactions. They can be implemented by inheriting the generic actuator class and then re-writing its run() method.

7.2.4 Conclusion

In this example we have discussed using the reactive system approach to detect and resolve the network partitioning problem. We use sensors/actuators to implement the failure detecting and notifying mechanism. The DMM in each subnet is the center for failure handling and coordination among replication managers. From the system architecture we can see that the DMM separates with the sensors and actuators, that is, the DMM stays the same no matter what sensors and actuators are, and vice versa. Hence if we want to change the system policies or sensors and actuators, we do not need to care about others because they do not have impacts on each other. The system fault-tolerant capability can be obtained from this architecture due to its continuing work even in the presence of DMMs or sensors/actuators failures or being changed. Compared with other strategies for partitioning problem, the reactive system method has the lower overheads and flexible system architecture.

7.3 The Web-based Teamwork Support System

The last application example in this thesis is the Web-based teamwork support system. Teamwork is a key feature in any workplace organization. With the development of network technologies, especially in the recent years Internet and
World Wide Web technologies, the computer-mediated teamwork is becoming increasingly important [Gorton 1996]. In this computer era, many tasks can be carried out by team members, who may be physically dispersed, cooperatively on Internet via the Web. Now there is a great deal of importance for the research into Web-based teamwork support for coordinating team members to get the real work done in a distributed environment.

Figure 7.7 depicts a task in a teamwork system, in which a process, project or task, generally speaking, is normally composed of sub-tasks which are partially ordered. By partially ordering, it means that a sub-task can only start when its previous sub-tasks have been completed. Normally, in a teamwork support system, 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. Hence, task-oriented technology is management-centred to facilitate project management focusing on coordination.

![Diagram of partial ordered sub-tasks]

Figure 7.7: The partial ordered sub-tasks

A teamwork support system in a distributed environment can be viewed as a reactive system because, for all team members in the system, they not only need to get their individual work done but need to communicate and collaborate with each other, i.e. there are interactions between the team members. This is the key feature of a reactive system. Therefore, we can use the reactive system concepts to model the teamwork support system. In the following sections we will discuss
Chapter 7. Applications

how to use the reactive system approach to solve the coordination problem in the Web-based teamwork system.

7.3.1 The Coordination of the Teamwork Support

The client-server architecture of a Web-based teamwork support system can be shown in Figure 7.8 [Yang 1998]. It includes clients as front-ends using local Web servers and tools, centralized servers with tools, and supporting tools such as databases as back-ends. The centralised server site plays the key role for management or coordination of a task. A Java application for a particular teamwork-oriented task runs at the server site as a daemon all the time serving for the entire life-span of the task. All task coordination related information is stored in a database repository which is accessed by the Java application for the task and the file system is also used to store centralized information such as documents. With the Java database connectivity (JDBC) facility, the Java application can use a variety of database systems without changing the code as long as such database drivers are available at the server site. Basically, at the client site, only an appropriate Web browser is required and no other particular software needs to installed since each team member uses Web pages on Internet and Java applets downloaded from the server site on-the-fly to carry out the sub-tasks allocated.

Figure 7.8: Information flow for supporting teamwork
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 [Yang 1998]. One promising method to implement such a mechanism may be that the server side, which realizes the coordination, provides every team member with a to-do list via an E-mail actively and also within the applet passively when a team member connects to the server with a correct identification. Once a team member has finished a sub-task on the to-do list, the notification should be made to the server such as simply by clicking a button on the Java applet. Then, at the server side, the appropriate coordination for process control can be adjusted to generate updated to-do lists for related team members.

At the client side, it is simple to deal with the to-do list and notification since they are mainly the work managed by the server. However, 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 environment, which often has restrictions for doing so. To solve this problem we embed the reactive system architecture into the teamwork support system and use the sensor/actuator mechanism to pass information between the server and team members for coordination, as depicted in Figure 7.9.

![Figure 7.9: The architecture of the teamwork coordination](image-url)
This figure shows the structure of a teamwork support system modelled with the reactive system concepts, where at the server site, a DMM class is embedded in the Java server and will run all the time for a teamwork-oriented task; it subscribes to the sensors and waits for the connections from client sites. At the client site, sensors are attached to the Java applet for each team member to obtain their states. The state information is sent to the DMM, and then the DMM reacts to the state events by using the actuators to change the states of the team members.

### 7.3.2 Implementation

The DMM, sensors and actuators used in the teamwork support system can be implemented by inheriting from the generic Java reactive system package developed earlier and re-writing the corresponding `run()` methods.

#### 7.3.2.1 Policies and decisions

The DMM in the teamwork support mainly has the following functions: First, it establishes a connection with the sensors and then listens to their connection requests; Second, when a team member connects to the server with a correct identification, the DMM will make decisions to decide a to-do list that lists the sub-tasks which can be started and send it to the team member. Third, once a team member has finished a sub-task, a notification will be made to the DMM through the sensor and then it makes the decision to generate the updated to-do lists and sends them to related team members.

In a teamwork support system, one important decision for the project management is that, when a sub-task can be started? or what conditions will be needed for starting a sub-task? The DMM has to determine the policies for the commencement of every sub-task. According to the policy, which may involve in other sub-tasks’ states, a sub-task can only be started when its policy condition is satisfied. From Figure 7.8, we know that the DMM will make decisions to determine whether a sub-task can be started on the basis of the related clients’
states received from the sensors which are attached to each client applet. When a policy condition for a sub-task is satisfied, the DMM will add this sub-task to a to-do list and send it to the related team member. Once a team member has finished a sub-task on the to-do list, the notification also will be reported to the DMM through the sensor. 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. This is the coordination mechanism for teamwork support using the reactive system model. The detailed decision making process can be referred to [Chen 1999].

7.3.2.2 Sensors/Actuators

Sensors used in the teamwork support system are timing sensors, i.e., they periodically report to the DMM about the states of clients. Each applet at client sites will create a sensor thread to monitor its state and report it to the subscriber (DMM) using the dedicated connection established between the sensors and the DMM. Sensor objects run and listen to the applet’s events all the time and report to the DMM periodically.

The actuators are simple and they are embedded in the sensors to receive the decisions from the DMMs and then instruct the applets to change their states.

7.3.3 Conclusion

The reactive system approach used in the Web-based teamwork support system provides the capability of fault-tolerance and flexible system architecture, even in the presence of partitioning failure. In the system it separates the DMM with the sensors and actuators, that is, the DMM stays the same no matter what sensors and actuators are, and vice versa, so that the flexibility and capability of fault-tolerance can be obtained from the system continuing work in the case of failures, or the DMM or sensors/actuators being changed. This is the advantage of the reactive system architecture and it can improve the system performance and make the system maintenance easier.
7.4 Summary

In this chapter we have presented three application examples to demonstrate the performance and potential benefits of the reactive system model. The first two examples discuss a distributed replication database system with site failure and partitioning failure scenario respectively. The third application addresses the Web-based teamwork support system.

Fault-tolerance is a very important feature of distributed systems. The reactive system model developed in the thesis has a broader application perspective. Three examples presented in this chapter are built based on this model. They have illustrated the flexible system architectures and the fault-tolerant capabilities, even in the case of site failures or partitioning failures. The advantage of the reactive system model is the separation of policies and mechanisms. In the three application examples, DMMs stay the same no matter what changes sensors and actuators have, and vice versa. This feature improves the system performance and makes the system fault-tolerant and easy to maintain. The system with this architecture can also meet constant changes in users’ requirements and its environment.
Chapter 8

Conclusion and Future Work

This chapter outlines the summary, the main contributions of the thesis, and addresses what can be improved in the future.

8.1 Summary of the Thesis

The focus of the research reported in this thesis is on supporting the development of distributed and fault-tolerant computing systems. The aim of the thesis is to provide a model based on the reactive system concepts for building better fault-tolerant computing applications by separating fault-tolerant computing policies from fault-tolerant mechanisms.

The reactive system concepts were introduced by D. Harel and A. Pnueli to describe a system that is supposed to maintain an ongoing interaction with its environment, rather than to produce some final result on termination. Most reactive systems are highly concurrent and distributed systems. They are generally composed of concurrent communicating and executing components or sub-processes, either for geographical requirements, or for safety and fault tolerance criteria. Concurrency gives rise to problems that are quite different from the ones sequential systems present, and they form the most difficult tasks faced
Chapter 8. Conclusion and Future Work

in the development of reactive systems.

From these concepts, we abstracted a generic reactive system architecture which consists of three layers: policies, mechanisms and applications. The reactive system uses decision making managers (DMMs) to implement policies regarding to the control of the applications. Mechanisms layer is implemented by using sensors/actuators. The major advantage of this architecture is that it separates policies from mechanisms. This advantage leads to a flexible software structure and has a great significance in developing fault-tolerant computing systems since it can separate fault-tolerant computing policies from its mechanisms.

The generic reactive system architecture is a component-based architecture which is composed of sensors, DMMs and actuators. To model this component-based architecture we introduced the actor and agent concepts to achieve this goal. An actor represents an active object that has a clearly defined purpose, and it may have its own execution thread and thus can operate concurrently with other active objects in its domain. Therefore, the actors can reflect the reactive nature of components in reactive systems. However, a DMM has some intelligence, and it is more like an agent. An agent is a software entity that can behave autonomously on behalf of its owner and it is an actor with intelligence. To better describe the nature of the DMM behavior, we use agents to model the DMMs in reactive systems. Therefore, we consider that a reactive system is a distributed computing system consisting of multiple DMM agents and sensor/actuator actors. This combination of actor and agent models has a good description of the reactive behaviors and has strong semantics, and increases the system modularity as well.

To ensure the correctness of the reactive system model, we have implemented the group communication mechanisms between the reactive components. The atomic multicast communication is invoked to provide the reliable communication and guarantee the complete communication, i.e., group members either all receive the message sent from a sender, or none of them receives. On the other hand, when different messages are sent to the different group members
concurrently, their arriving orders may be different. Therefore three ordering constraints, i.e. FIFO, total ordering and causal ordering, are introduced to the message sets in the system. Together with the multicast atomicity protocol they have been implemented to guarantee the correctness of the reactive semantics.

The implementation of the reactive system model involves implementing various reactive control protocols, such as communication, temporal constraints and ordering constraints, etc. The communication between DMM, sensor and actuator can be implemented with two patterns: multicast datagram and stream-based communication, therefore, these entities are implemented with both multicast datagram and stream-based communication methods. Using various features of the Java language, DMMs, sensors and actuators are also implemented as both single and multi-threaded entities which can be distributed over the networks. The group communication mechanisms are implemented in the encapsulations of these entities.

The evaluation of the reactive system model has been conducted based on the performance of the system with different configurations and running in different distributed environments. The performance study is conducted by measuring the average response time (ART) of the system to outside inputs. This study has shown that the reactive system model designed in the thesis running in a distributed environment is more effective than it running in a centralized computer. We also introduced three application examples in the thesis to further demonstrate the potential benefits of the reactive system model. Our reactive system model has a broader application perspective. In three examples, the model provided a flexible system architecture and improved the system performance, even in the presence of site failures or partitioning failures. Due to the separation of policies and mechanisms, DMMs stay the same no matter what changes sensors and actuators have, and vice versa. This separation makes the system fault-tolerant and easy to maintain. Also, the system with this architecture can meet constant changes in users’ requirements and its environment, thus the system performance can be improved.
8.2 Main Contributions

In general, the research reported in this thesis has a significant impact on the distributed and fault-tolerant computing theory and application. The results of the research can be directly applied into many applications. Specifically, the research has achieved the following goals:

8.2.1 Modeling and Design of Reactive Systems

From the reactive system concepts, we proposed a generic reactive system architecture model, which consists of three layers: policies, mechanisms and applications. The major advantage of this architecture model is the separation of policies and mechanisms. This advantage leads to a flexible software structure and has a great significance in developing fault-tolerant computing systems because it can separate fault-tolerant computing policies from its mechanisms.

The generic reactive system architecture is composed of decision making managers (DMMs), sensors and actuators. After proposing this model, we used the actor and agent concepts to formally build and design its contents. Actors can reflect the reactive nature of components in reactive systems. However, a DMM has some intelligence, and it is more like an agent. To better describe the nature of DMMs’ behaviors, we use agents to model DMMs in reactive systems. Therefore, we formally model a reactive system as a distributed computing system consisting of multiple DMM agents and sensor/actuator actors. This combination of actor and agent models has a good description of the reactive behaviors and has strong semantics, and increases the system modularity as well. With this modeling and design, we can understand the reactive behaviors easily and prove the safety and liveness properties of the system. These properties ensure that the system always run correctly and is fault-tolerant.
8.2.2 Generic Java Reactive System Package

After modelling the generic reactive system, we have implemented it as a software package using the Java programming language. The generic Java reactive system package can be applied in many applications directly. Specifically, it includes the following classes:

- A generic Java decision making manager class that implements some fault-tolerant policies. In particular, two specific DMMs with replication-based fault-tolerant policies have been implemented using the generic DMM: primary-backup replication policies and active replication policies.

- A generic Java sensor class that can monitor and report the states of distributed objects to its subscribers. Using such a generic sensor, three types of specific sensors have been implemented: event sensor, timing sensor and polling sensor.

- A generic Java actuator class that can change the states of applications. Using such a generic actuator, two types of specific actuators can be implemented: warm start-up actuator and cold start-up actuator.

- A group communication sub-system that offers mechanisms of reliable multicasting, ordered message delivery, and consistent group membership view. These features are essential for the coordination among components (such as sensors, actuators, DMMs, and application objects) in a reactive system. In particular, two levels of services are provided by this sub-system: reliable and orderly message delivery primitives and group view services.

8.2.3 Performance Analysis

We have conducted the performance study to evaluate the generic Java reactive system package after the implementation. In the evaluation, we measured the average response time of the system under different configurations to compare
Chapter 8. Conclusion and Future Work

the effects of different factors that have impacts on the system, such as communication methods, network environments, and various constraints. The study has shown that the reactive system model designed in the thesis running in a distributed environment is more effective than it running in a centralized computer.

We also introduced three application examples in the thesis to further demonstrate the potential benefits of the reactive system model. The first two examples are discussing a distributed replication database system with site failure and partitioning failure scenario respectively. The third application is the Web-based teamwork support system. The distributed replication database system is applied in Deakin University covering multiple campuses to maintain some student, staff and course information. The Web-based teamwork support system is applied in Deakin University as well to coordinate the cross-campus course development task. The goal of these systems is to guarantee that the systems provide continuous services even in the presence of a server failure, a site failure, or a network partitioning failure. These applications have demonstrated the direct applications of our research results.

8.3 Future Work

The reactive system model developed in the thesis can be applied into many applications directly. However, it can be improved in the following aspects:

- **The language level** -- A reactive system interface definition language (RSIDL) and its compiler. The RSIDL can allow a user to specify parallelism, synchronization, fault-tolerance, and other special issues between system components (such as sensors, actuators, DMMs, and application objects) at a higher level. The RSIDL compiler translates an interface written in RSIDL into client, server and client-server stubs, making use of the services provided by the group communication sub-system. Stubs are integrated into various system components according to the roles (client, server, or client-server)
these components are to play. The RSDL and its compiler can be based on work on remote procedure calls [Zhou 1995(b)] [Zhou 1995(e)].

- Reliability evaluation. As computer systems become more and more complex, reliability modeling and evaluation is now an extremely hard task [Siewiorek 1992]. Besides various hardware failures, software failures exist in some applications. The reactive system model developed in the thesis can be used in both the hardware and software fault-tolerant computing. Software faults are caused by improper specification, bad design, or inadequate implementation [Mili 1990]. For example, DMMs, sensors or actuators used in specific applications may fail due to bad design or inadequate implementation. How to evaluate their reliabilities remains an important task.

This thesis has just developed the reactive system model for building fault-tolerant distributed applications. It hasn't included any toolkit to assist the construction of specific applications. One of the directions of the further research on this topic is to provide such a toolkit that includes the automatic construction of various DMMs, sensors and actuators, the RSDL, and the evaluation of reliability as well.
References


References


References


References


References


References


References


References


171
References


References


References


References


References


