I am the author of the thesis entitled *The Development of DAL and DAPL Languages for Building 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: ROBERT ALAN DEW

Signed: Signature Redacted by Library

Date: 12/7/2003
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>Justin D. Pierce</td>
<td>Signature Redacted by Library</td>
<td>18.8.4</td>
</tr>
</tbody>
</table>

Deakin University Library

3 3217 0095 8996 2
The Development of
DAL and DAPL Languages
for Building
Distributed Applications

by
ROBERT DEW, B.Sc.(Hons), M.Sc.

Submitted in fulfilment of the requirements for the degree of
Doctor of Philosophy
Deakin University (December, 2002)
DEAKIN UNIVERSITY
CANDIDATE DECLARATION

I certify that the thesis entitled The Development of DAL and DAPL Languages for Building Distributed Applications

submitted for the degree of Doctor of Philosophy

is the result of my own work and that where reference is made to the work of others, due acknowledgment is given.

I also certify that any material in the thesis which has been accepted for a degree or diploma by any other university or institution is identified in the text.

Full Name: ROBERT ALAN DEW
Signed: .................................................................
Date: 12/7/2003
ACKNOWLEDGEMENTS

I sincerely thank my supervisors, Peter Horan and Andrzej Goscinski, for their advice, support and keen interest during this research; and my parents, Ivon and Sylvia Dew, for their direct and indirect support of education.
## Contents

Abstract .................................................. 1

1 Introduction ............................................. 3
   1.1 Background ......................................... 3
      1.1.1 Tightly Integrated Connectivity ............... 3
      1.1.2 Implied Connectivity .......................... 4
      1.1.3 No Support for Specifying Connectivity ....... 5
      1.1.4 Explicit Connectivity .......................... 5
   1.2 Research Scope ...................................... 5
   1.3 Aims of this Research ............................... 6
   1.4 Thesis Overview ..................................... 7

2 Literature Review ...................................... 9
   2.1 Characteristics of Distributed Programming ....... 9
   2.2 Distributed Programming Languages and Systems .... 17
      2.2.1 Object Oriented Parallel Programming Languages 19
      2.2.2 Control Flow and Data Flow Languages ......... 20
      2.2.3 Functional Languages ............................ 22
      2.2.4 Process Oriented Languages ..................... 22
      2.2.5 Shared Memory Systems ........................... 23
      2.2.6 Library and Tool Systems ....................... 24
   2.3 Project Requirements ................................ 26
      2.3.1 DAS Requirements ................................ 27
      2.3.2 Distributed Application Requirements .......... 28
      2.3.3 DAL and DAPL Programming Language Requirements 30
   2.4 Conclusion ........................................... 31

3 Distributed Applications: Logical Design .......... 32
   3.1 Connectivity of Application Processes ........... 33
      3.1.1 Data Types ...................................... 33
      3.1.2 Communication Channels ......................... 33
      3.1.3 Processes ........................................ 34
      3.1.4 Process-Buffer Pair ............................. 34
      3.1.5 Interfaces for Modular Design ................... 34
      3.1.6 Computer Names .................................. 34
      3.1.7 Creating Processes in Sequence ................ 35
Contents

3.2 Application Processes 35
  3.2.1 Operational Semantics 35
  3.2.2 Buffers 37
  3.2.3 Outputs 38
  3.2.4 Functions 38

3.3 Summary 39

4 Specifying Distributed Applications Using DAL and DAPL 40
  4.1 Distributed Application Diagram 40
    4.1.1 Example Diagram of a Distributed Application 43
  4.2 DAL: Distributed Application Language 44
    4.2.1 DAL Specifications 44
    4.2.2 Validating Connectivity 47
    4.2.3 Examples of DAL Specification 48
  4.3 DAPL: Distributed Application Process Language 51
    4.3.1 DAPL Programs 51
  4.4 DAS Library User Functions 60
    4.4.1 DA_send() 60
    4.4.2 DA_remove() 61
    4.4.3 DA_head() 62
    4.4.4 DA_free() 62
    4.4.5 DA_exit() 63
    4.4.6 DA_exitapp() 63
  4.5 Example: Distributed Fibonacci 64
    4.5.1 Distributed Fibonacci: Diagram 64
    4.5.2 Distributed Fibonacci: DAL Specification 65
    4.5.3 Distributed Fibonacci: DAPL Programs 66
    4.5.4 Distributed Fibonacci: Output 69
  4.6 Conclusion 70

5 Hierarchical Specifications of Distributed Applications Using DAL 73
  5.1 Graphical Notation for Hierarchical DAL Specifications 75
    5.1.1 Parent-Child Relationship 75
    5.1.2 Interface 75
    5.1.3 Example: Fibonacci Sequences 77
    5.1.4 Converting Hierarchical to Non-Hierarchical
          DAL Specifications 79
    5.1.5 Multiple Re-use of a DAL Specification 80
5.2 The DAI. Language for Hierarchical DAI. Specifications 81
  5.2.1 Interface Buffers 82
  5.2.2 Interface Outputs 83
  5.2.3 Example: Interface Buffers and Outputs 84
  5.2.4 DAL Specification: Requires, Machines and Creation Clause 85
5.3 Example: Fibonacci Sequences 86
5.4 Conclusion 92

6 DAS: The Distributed Application System 94
  6.1 DASH: User Interface to DAS 95
    6.1.1 Host Name Management 95
    6.1.2 Compilation 95
    6.1.3 Distributed Application Management 100
    6.1.4 Internal DASH Commands 103
  6.2 Compilation System 104
    6.2.1 The DAI. Parser 106
    6.2.2 The Canonical DAL Generator 110
    6.2.3 The Canonical DAL Integrator 111
    6.2.4 The DAPI. Parser 111
    6.2.5 The C Code Generator 114
  6.3 Management of Distributed Applications 122
    6.3.1 Application Management: Pre-instantiation 122
    6.3.2 Application Management: Post-instantiation 123
    6.3.3 Application Management: During Run-time 125
  6.4 Summary 126

7 Implementation of Distributed Applications 127
  7.1 DAL Language Implementation 128
  7.2 DAPI. Language Implementation 130
    7.2.1 Implementation of Process Buffers 131
    7.2.2 Implementation of Process Outputs 135
    7.2.3 Sending Data: Internal Data Packet 140
  7.3 DAS System Implementation 141
    7.3.1 Distributed Application Shell (DASH) Implementation 141
    7.3.2 DAS Compilation System Implementation 143
    7.3.3 DAS Data Structures 144
    7.3.4 DAS Messages 148
  7.4 Conclusion 160
8 Testing the DAS System and Distributed Applications 161
  8.1 Tasks to Achieve Research Aims 162
  8.2 Testing Fundamental Connectivity Arrangements 163
    8.2.1 Test 1: Single Output from One Process to Single Buffer 165
    8.2.2 Test 2: Two Outputs from One Process to a Single Buffer 165
    8.2.3 Test 3: Two Outputs from Two Processes to a Single Buffer 166
    8.2.4 Test 4: Two Outputs for Bidirectional Data Transfer 167
    8.2.5 Test 5: Two Outputs from One Process to Two Buffers 168
    8.2.6 Test 6: Two Outputs from Two Processes to Two Buffers 169
    8.2.7 Test 7: Sending Arrays of Integers 170
    8.2.8 Test 8: Sending Arrays of Integers and Strings 171
    8.2.9 Test 9: Sending Arrays of Structures 172
  8.3 Testing Fundamental Hierarchies 174
    8.3.1 Test 1: Parent and Child Relationship 176
    8.3.2 Test 2: Parent, Child and Grandchild Relationships 177
    8.3.3 Test 3: Parent of Several Children 179
    8.3.4 Test 4: Single DAL Specification for Several Children 180
  8.4 Managing Multiple Applications 183
  8.5 Conclusion 184

9 Conclusion 185
  9.1 Original Results 185
  9.2 Future Work 187
  9.3 Summary 188

References 189

A DAL Grammar 197
B DAPL Grammar 199
C Structures Used by the DAL Parser 202
D Structures Used by the DAPL Parser 206
E Identification Numbers of Fundamental Types 209
F Listing of the Header File das_structs.h 210
G Testing Multiple Applications 217
H DAL Specifications and DAPL Programs: Testing Connectivity 229
I DAL Specifications and DAPL Programs: Testing Hierarchy 249
J DAS Generated C Code: Example 270
## List of Figures

3.1 Two applications and the DAS process distributed on four workstations. 32
3.2 The logical view of a process in a distributed application. 36
4.1 Graphical components of a distributed application diagram. 41
4.2 A diagram called example_1 of two processes. 41
4.3 A diagram of two processes and their associated buffers. 42
4.4 Distributed application diagram depicting the connectivity of two processes. 42
4.5 Two outputs named y and z are associated with the same buffer buf_B. 43
4.6 A small application showing how processes can be connected. 44
4.7 An extended DAL specification of the diagram presented in Figure 4.4. 48
4.8 A DAL specification of the application diagram presented in Figure 4.6. 49
4.9 A DAL specification with machines and creation clauses. 50
4.10 The general format of a DAPL program. 52
4.11 Example structure definitions for a DAPI program. 53
4.12 Example of a buffers clause of a DAPL program. 54
4.13 A generic form of a DAPL buffer statement. 56
4.14 Example outputs clause of a DAPL program. 57
4.15 A generic form of a DAPL program consisting of one output. 58
4.16 A distributed application diagram of the Fibonacci application. 65
4.17 A DAL specification of the Fibonacci application. 65
4.18 A DAPL program for the process display_Fibonacci. 67
4.19 A DAPL program for the process compute_Fibonacci. 68
4.20 A transcript containing DASH commands to create the Fibonacci application, application output, and output generated when the application terminates. 70
4.21 A DAL specification that represents the minimum amount of code required to specify an application of two connected processes. 71
4.22 A distributed application diagram of the DAL specification of Figure 4.21. 71
5.1 A distributed application diagram depicting a parent-child relationship as a pair of dashed lines. 74
5.2 A distributed application diagram containing an interface output c. 76
5.3 A distributed application diagram containing interface buffers x and y. 76
5.4 A diagram of the Fibonacci sequences application. 78
5.5 A distributed application diagram where an application that displays one sequence of Fibonacci numbers is used in the new application.

5.6 A non-hierarchical version of the hierarchical diagram in Figure 5.5.

5.7 A diagram of a hierarchical DAL specification.

5.8 Interface statements for the application Fibonacci from Figure 5.5.

5.9 DAL statements for the process Fibonacci_sequence.

5.10 The DAL specification for application Fibonacci_sequences.

5.11 The DAL specification for application Fibonacci.

5.12 A DAPL program for process display_sequence.

5.13 A DAPL program for process create_sequence.

5.14 A DAPL program for process compute_Fibonacci.

6.1 A dataflow diagram of the compilation system.

6.2 A distributed application diagram depicting a parent-child relationship as a pair of dashed lines.

6.3 The C code generator and associated input and output files.

6.4 The makefile for the application called example_app.

6.5 Createfile for the application called example_app.

6.6 Entity-relationship diagram of the DAS system, DAL specifications, DAPL programs, applications, processes and outputs.

7.1 The non-hierarchical DAL specification produced by flattening the hierarchical DAL specification in Figure 5.10 and Figure 5.11.

7.2 Several logical communication channels between two processes are implemented using one pair of sockets to form one communication channel.

7.3 Pseudo code of the dash() function.

7.4 Pseudo code of the dash_compile() function.

7.5 Pseudo code of the das_create() function which is used to create the DAS system structure.

7.6 A diagram depicting the relationships among the DAS system structure, das, and structures required to store information about three applications. $ denotes a das_DAL_application structure, # denotes a dal_application structure, and + denotes a dapl_program structure.

7.7 Relationships among the DAS system structure and structures for storing information about application instances.

7.8 Pseudo code for creating and running a distributed application.

7.9 Pseudo code for terminating a distributed application.

7.10 The sequence of four messages required to create an output.

7.11 The sequence of four messages required to close an output.
List of Figures

7.12 The sequence of messages required to create an application process. 156
7.13 The sequence of messages required terminate an application process. 157
7.14 The sequence of messages required during a process move operation. 159
8.1 A distributed application consisting of N processes connected in an assembly line fashion. 161
8.2 An application for sending an integer from one process to another. 165
8.3 An application for sending two integers from one process to one buffer. 166
8.4 An application for sending two integers from two processes to one buffer. 167
8.5 An application to test bidirectional data flow. 167
8.6 An application to send integers from one process to two buffers within another process. 168
8.7 An application to send integers from two processes to two buffers within another process. 169
8.8 An application to send arrays of integers. 170
8.9 An application to send arrays of different types. 172
8.10 An application to send several arrays structures. 173
8.11 A parent/child relationship between two DAL specifications. 175
8.12 A child/grandchild relationship between two DAL specifications. 175
8.13 Relationships between a parent and its two children. 175
8.14 Relationships between a parent of two twins. 176
8.15 An application to display sequences of Fibonacci numbers. 178
8.16 An application to display sequences of Fibonacci numbers. 180
8.17 An application of a parent node with multiple children. The same DAL specification is used for each child. 181
List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>6.1</td>
<td>DASH commands.</td>
<td>96</td>
</tr>
<tr>
<td>6.2</td>
<td>Components of generated C programs.</td>
<td>116</td>
</tr>
<tr>
<td>8.1</td>
<td>Summary of testing the 9 fundamental connectivity arrangements.</td>
<td>164</td>
</tr>
<tr>
<td>8.2</td>
<td>Summary of testing the fundamental hierarchies.</td>
<td>176</td>
</tr>
</tbody>
</table>
Abstract

A common characteristic among parallel/distributed programming languages is that the one language is used to specify not only the overall organisation of the distributed application, but also the functionality of the application. That is, the connectivity and functionality of processes are specified within a single program.

Connectivity and functionality are independent aspects of a distributed application. This thesis shows that these two aspects can be specified separately, therefore allowing application designers to freely concentrate on either aspect in a modular fashion. Two new programming languages have been developed for specifying each aspect. These languages are for loosely coupled distributed applications based on message passing, and have been designed to simplify distributed programming by completely removing all low level interprocess communication.

A suite of languages and tools has been designed and developed. It includes the two new languages, parsers, a compilation system to generate intermediate C code that is compiled to binary object modules, a run-time system to create, manage and terminate several distributed applications, and a shell to communicate with the run-time system.

DAL (Distributed Application Language) and DAPL (Distributed Application Process Language) are the new programming languages for the specification and development of process oriented, asynchronous message passing, distributed applications. These two languages have been designed and developed as part of this doctorate in order to specify such distributed applications that execute on a cluster of computers. Both languages are used to specify orthogonal components of an application, on the one hand the organisation of processes that constitute an application, and on the other the interface and functionality of each process. Consequently, these components can be created in a modular fashion, individually and concurrently.

The DAL language is used to specify not only the connectivity of all processes within an application, but also a cluster of computers for which the application executes. Furthermore, sub-clusters can be specified for individual processes of an application to constrain a process to a particular group of computers. The second language, DAPL, is used to specify the interface, functionality and data structures of application processes. In addition to these languages, a DAL parser, a DAPL parser, and a compilation system have been designed and developed (in this project). This compilation system takes DAL and DAPL programs to generate object modules based
on machine code, one module for each application process. These object modules are used by the Distributed Application System (DAS) to instantiate and manage distributed applications.

The DAS system is another new component of this project. The purpose of the DAS system is to create, manage, and terminate many distributed applications of similar and different configurations. The creation procedure incorporates the automatic allocation of processes to remote machines. Application management includes several operations such as deletion, addition, replacement, and movement of processes, and also detection and reaction to faults such as a processor crash. A DAS operator communicates with the DAS system via a textual shell called DASH (Distributed Application SHell).

This suite of languages and tools allowed distributed applications of varying connectivity and functionality to be specified quickly and simply at a high level of abstraction. DAL and DAPL programs of several processes may require a few dozen lines to specify as compared to several hundred lines of equivalent C code that is generated by the compilation system. Furthermore, the DAL and DAPL compilation system is successful at generating binary object modules, and the DAS system succeeds in instantiating and managing several distributed applications on a cluster.
1 Introduction

A distributed application is generally viewed as a computer program consisting of several computational elements which execute on several computers connected by a network. Many elements can execute concurrently on a single processor computer and others can execute on processors of a multiprocessor computer. In addition to execution, computational elements may communicate with each other using message passing, shared memory, or distributed shared memory. Therefore, the specification of a distributed application must explicitly (or implicitly for shared memory or DSM) contain specifications for both computational elements and communications.

1.1 Background

This research is based on being able to explicitly specify the connectivity of a distributed application using simple notation, and independently from functionality. For example, if process P can send messages to process Q via communication channel C, then notation to specify this connection consists of identifiers of the two processes and the channel, and the direction of message transmission. Several notations can be easily formed, e.g., "P to Q via C", "P,Q,C", "P > C > Q", "P C Q", or "C(P,Q)". Following the completion of specifications for both connectivity and functionality, these specifications can be used by a compilation system to generate executable object modules. A run-time system can then instantiate a distributed application by using these object modules to create processes remotely.

Existing languages and tools used for developing distributed applications may:

- tightly integrate connectivity and functionality within a specification (e.g., C++, Orca and PVM);
- imply connectivity (e.g., Linda, SR and CORBA);
- provide no support for specifying connectivity (e.g., MPI); or
- explicitly specify connectivity (e.g., HeNCE, CODE and VPF).

1.1.1 Tightly Integrated Connectivity

Network programming using C++ involves writing advanced low level code to create communication channels between two processes. In addition, such C++ code must contain statements to create the second process. For applications of many processes the connectivity of such C++ programs is tightly bound with the program’s functionality, and it is a complex task to separate such connectivity from functionality because the number of processes and sockets is not explicit, i.e., the overall structure of the application is unknown.
Chapter 1. Introduction

Orca is an object-based language for distributed applications. An Orca application commences with one process, and for parallelism it must use a fork statement to create additional processes at run-time [Bal (1990)]. Therefore, connectivity of an Orca application is unknown at the time of specification, and implicit in the functionality of the application.

PVM (Parallel Virtual Machine) is a library of low level tools [Beguelin et al (1993a)] such as functions for task creation, buffer creation, packing data into buffers, and message passing. The PVM library function to send a message requires a buffer of data, a task identification number of the sender, and a message identification number. The receive function requires the identification numbers of the sender and sent message. The connectivity of PVM tasks is heavily reliant on not only send and receive function calls throughout the code, but also programmer supplied identification numbers. It is therefore tightly integrated within the functionality of a PVM parallel program.

1.1.2 Implied Connectivity

Distributed applications based on shared memory or distributed shared memory such as Linda [Ahuja et al (1986)] and SR [Olsson et al (1992), and Andrews and Olsson (1993)] essentially permit a chunk of virtual memory to be within the scope of one or more processes. Linda is a shared memory language where only one chunk of memory (a tuple space) is shared by all processes. Any process can asynchronously add a tuple (data) into the tuple space, and any process may remove a tuple from the tuple space by matching the tuple’s format. The connectivity of a Linda application is implicit because all processes have access to the tuple space. However, such connectivity is only between process and tuple space, there is no direct connectivity from one process to another.

SR is a concurrent language that uses shared memory in a manner similar to Linda. Whereas Linda shares one tuple space among all processes, SR shares several message queues among all processes. Like Linda, there is no direct connectivity between SR processes with respect to message queues. However, SR also provides a remote procedure call (RPC) mechanism, by which means connectivity can be specified between two processes. But this connectivity is RPC dependent and, consequently, is tightly integrated within the functionality of the application.

A popular product is CORBA (Common Object Request Broker Architecture) which is a specification of an ORB (Object Request Broker) for providing client-object communications. The ORB delivers messages from clients to objects, and returns results to clients [Vinoski (1997)]. Clients and objects are both CORBA components, and a component takes a client role when requesting a service of another
component which performs the object (server) role. However, a client component must provide the ORB with an object reference\(^1\) of the server object as part of a request. Object references are only available to a client during run-time either by object creation by the client, or as the result of a request by the client to a directory service. Therefore the connectivity of a CORBA application is intrinsically implied.

1.1.3 No Support for Specifying Connectivity

MPI (Message Passing Interface) is a popular message passing library, but not a system for specifying or developing complete distributed applications [Dongarra et al (1996)]. This is because MPI lacks both process management and tools for application development.

1.1.4 Explicit Connectivity

HeNCE, CODE and VPE are environments for graphically specifying parallel applications. HeNCE [Beguelin et al (1993)] graphs contain nodes and arcs. A node represents a procedure (C like function) and only commences execution when all nodes that are connected by an incoming arc to this node complete executing. This is similar to invoking a procedure, i.e., a procedure commences when all of its input parameters have been evaluated. Arcs represent data flow from one node to another, or control flow (repetition and selection constructs).

CODE is very similar to HeNCE [Browne et al (1994)]. Nodes are procedures, but arcs represent data flow only. Nodes within CODE and HeNCE only communicate at the beginning and end of a computation [Newton et al (1995)]. This communication model is quite restrictive because it lacks the capability whereby a computational element can send messages at any time during a computation. HeNCE and CODE have serious problems [Newton and Dongarra (1995)] because the graphs are complicated, awkward and large.

Like HeNCE and CODE, VPE [Newton and Dongarra (1995)] graphs contain nodes and arcs. However, a VPE node represents a process, and an arc represents a communication channel having a port at both ends. VPE messages are not typed and may consist of an unknown arrangement of types [Newton et al (1995)].

1.2 Research Scope

The objective of this research is to explore means for developing distributed applications that use message passing. A major part of the development of such applications is using facilities to specify the distribution of the application on a

---

\(^1\) A CORBA object reference uniquely identifies an object.
network of computers. These facilities allow the creation of specifications for distributed applications, compilation of these specifications forming an application of linked object modules, and parallel execution of these modules on the network.

The specification has two distinct parts. The first part is the specification of the application's structure in terms of which elements communicate with which other elements, i.e., the connectivity of computational elements. The second part is the specification of functionality of computational elements.

The separation of connectivity specifications from functionality specifications is important because it provides designers with greater freedom to produce applications. Designers can specify the structure of an application without affecting functionality. Furthermore:

- An independently specified structure can be treated as a framework. So an application can be developed by inserting computational elements in it. Therefore, applications can be updated by replacing computational elements with newer versions without considering or altering the application's structure.
- Producing high level specifications of connectivity means that low level code can be automatically generated from these specifications. Therefore, the application programmer need not be concerned with low level network programming such as UNIX sockets [Stevens (1990)].

1.3 Aims of this Research

The aim of this research is to develop a technology for efficient building of distributed applications consisting of concurrently running processes distributed on a network, and where communications is based on asynchronous message passing. Messages are typed arrays of data sent by one process and automatically placed into a buffer within the receiving process. The tasks to be solved to achieve this aim are:

- to specify applications such that connectivity specifications are independent from functionality specifications; that is, to enable separate specification of an application's structure from the specification of its functionality;
- to specify connectivity in a modular fashion to provide re-use of existing connectivity specifications;
- to build applications by combining specifications of functional components with connectivity specifications, and then compiling this combined specification to produce executable object modules;
- to distribute object modules on a network in order to execute a distributed application; and
- to manage several running distributed applications using a single process.
1.4 Thesis Overview

The development of distributed applications by specifying connectivity independently from functionality by using two new programming languages (DAL and DAPL) is presented in this thesis. In addition, the combination of such specifications can be compiled to produce executable distributed applications, and these applications can be managed by a single process called DAS (Distributed Application System). The DAL and DAPL languages, DAS run-time system, implementation details, and test results are presented within the following chapters.

Chapter 2 contains the literature review. The first section presents distributed programming characteristics such as multiprogramming, multicomputer, shared memory and message passing with relevant languages and systems such as Linda and PVM. These characteristics are not common to all languages and systems. For example, process communications in Ivy is based on virtual shared memory; and process communications in PVM is based on message passing and does not use shared memory. In addition, some characteristics are not common to all languages of a particular class. For example, ABCL, pC++ and Arjuna are object oriented languages. ABCL is based on distributed memory; pC++ is based on shared memory and distributed memory; and Arjuna use RPC communications. The second section of Chapter 2 contains a description of different kinds of language for distributed applications such as object oriented, process oriented, data flow and functional; shared memory systems such as Linda; and libraries such as CORBA, PVM, MPI and P4. The third section contains detailed requirements for this research project.

The logical design of the kinds of distributed application upon which this research is focused is presented in Chapter 3. These applications consist of communicating processes distributed over a network of loosely coupled computers. The logical design of such processes is presented in four sections. The way a process commences execution, sends data, and responds to incoming data is presented in the first section. Buffers are used to store incoming data. They are local to a process and discussed in the second section. The third section contains the logical view of communication channels. The sending process sends data by logically placing data onto a channel whereby the run-time system transfers and inserts this data into a buffer of the receiving process. Functionality of a process is presented in the fourth section. A process waits for incoming data to arrive into one of its buffers, and on arrival, automatically invokes a function associated with this buffer to process the data.

Details of the DAL and DAPL languages, library functions such as DA_send(), and examples of distributed application written in DAL and DAPL are presented in Chapter 4. It is shown in this chapter that distributed applications can be
specified as two independent parts. The DAL language is used to specify connectivity and the DAPL language is used to specify functionality. Chapter 5 presents the parts of the DAL language that support the specification of an application's connectivity in a hierarchical way. This also allows modular re-use of existing specifications within a hierarchical DAI specification.

Chapter 6 presents the Distributed Application System (DAS) in three sections. The first examines the user interface called DASH (Distributed Application SHELL) which allows an operator to specify a list of computer names to define the host network. It also describes commands to compile DAL specifications and DAPL programs, create applications, and terminate them. The compilation system is presented in the second section. It consists of several components that use DAL specifications and DAPL programs to generate object modules for creating a distributed application. These components include the DAL parser, the DAPL parser, an C code generator, and an C compiler. The third section presents what information is stored by the DAS system to create, run, and terminate distributed applications.

Chapter 7 presents the implementation of the DAL and DAPL languages, and the DAS system. The DAL and DAPL parsers are implemented using the flex and yacc compiler tools. They determine whether DAL specifications or DAPL programs are valid with respect to the DAL and DAPL languages respectively, and also extract and store information into several data structures to be used by the DAS system. DAPL programs contain declarations of buffers and communication channels for message passing. Implementation details of such buffers, channels, and internal data packets are also presented in this chapter. The final sections of Chapter 7 present implementation details of the DAS system, i.e., the DASH shell, the compilation system, data structures for storing data about applications, and the internal DAS messages to manage applications.

Chapter 8 contains details on testing the DAS system. Testing focuses on the ways processes may be connected. Such applications are specified, compiled and executed to test the DAS system. Expected results matched actual results for all tests. Chapter 9 is the final chapter whereby conclusions and future work are presented.
2 Literature Review

In previous work, modelling of real-time systems [Dew (1992)] using Petri nets and object oriented concepts produced a system for rapidly prototyping such real-time systems [Dew (1993) and Dew (1994)]. The notation by Ward and Mellor (1985) [Ward (1986), and Ward (1988)] was the basis for producing these prototypes. This work led to research on the actual specification and implementation of systems characterised by concurrent and parallel execution of computational elements. In brief, the outcomes of this research are two new languages (DAL and DAPL) for specifying distributed applications, associated parsers and language translators, and a system to manage such distributed applications running as a set of cooperating processes on a network of UNIX machines.

Distributed software systems can be designed and implemented using one or more techniques, and therefore a designer of distributed applications needs to be aware of such techniques. The following sections of this chapter present characteristics of distributed programming such as message passing, shared memory and distributed memory; methodologies, languages, and libraries for distributed programming such as object oriented, process oriented, abstractions like tuple-spaces, PVM and MPI; and the requirements for developing distributed applications as part of this project.

2.1 Characteristics of Distributed Programming

Different techniques of programming distributed application such as object oriented or functional, can be found in the literature. In addition, within each one of these techniques are different ways for computational elements to communicate. The computational elements vary too. Three taxonomies of distributed application programming are:

1. Distributed Programming Languages and Systems - the different kinds of languages and systems for developing distributed applications are object oriented, control flow and data flow, functional, process oriented, shared memory, libraries and tools.

2. Communication Models - communication models used for developing distributed applications are described within the literature and below. These are synchronous and asynchronous message passing, RPC, shared memory, distributed shared memory, and shared objects.

3. Computational Models - a variety of computational elements constituting distributed applications have been proposed. There are two main kinds: coarse-
grained and fine-grained. Such computational elements are processes, tasks (c.g. a PVM task), threads, procedures, functions, objects and object methods.

Some languages for distributed programming, as presented above, are based on different kinds of computational and communication models. In addition to these, other important concepts associated with programming distributed systems are presented within this section.

**Multi-programming.** Multi-programming is the concurrent execution of several independent programs on one processor [Ben-Ari (1990)]. For example, the UNIX operating system provides multi-programming.

**Shared Memory.** Shared memory is (presented as) a memory architecture or a memory segment that is directly accessible by more than one processor [Lafferty *et al* (1993), and Stevens (1990)]. Therefore, shared memory is within the scope of not only several processors, but also several processes running on such processors. In addition, synchronisation of processes accessing the same piece of shared memory is essential, otherwise data can be easily corrupted, e.g., one process reads (old) data from the memory before another process writes valid data to the memory. Two languages based on shared objects are Amber and Orca.

- An Amber program [Chase *et al* (1989)] is an object oriented subset of C++ that provides a shared-object virtual memory on a network of Topaz computers (shared memory multiprocessors). Amber code and read only data objects are replicated on all processors, i.e., the same code is placed on processors.
- The Orca language [Bal (1990)] uses shared data objects (instances of user defined ADTs) and the programmer does not have access to shared memory.

**Distributed Memory.** Distributed memory is a memory architecture in which each processor has its own local memory [Lafferty *et al* (1993)]. Therefore, a memory segment is within the scope of only one processor, and processes running on one processor cannot access memory belonging to another processor. Some languages such as ABCL, Charm and pC++ are purposely designed to suit shared and/or distributed memory systems.

- ABCL is an object oriented language for fine-grained parallel programming on the AP1000 distributed memory computer. An AP1000 computer consisted of 64 to 1024 processors, each with 16MB of RAM, and the memory is not shared [Taura *et al* (1993), and Yonezawa *et al* (1986), and Kale *et al* (1994)].
- Charm is an object based parallel programming language for MIMD parallel machines [Kale *et al* (1994a), and Kale *et al* (1994b)] consisting of hundreds of
processors, utilising shared memory or distributed memory. Charm++ is an object oriented version of Charm that is essentially C++, and where global variables are only permitted for objects that are explicitly created for sharing information [Kale and Krishnan (1993)]. An object in Charm++, called a Chare, has a unique ID, and therefore Charm++ supports global IDs, as well as global information sharing objects.

- pC++ is an SPMD object-oriented parallel programming language for both shared and distributed memory parallel systems [Bodin et al (1993), Bodin et al (1993), and Lee and Gannon (1991)]. Parallelism is based on applying an operation to several objects in parallel, these objects are distributed over the processors of massively parallel computers such as a CM-5 from Thinking Machines Corporation.

**Distributed Shared Memory.** Distributed shared memory (DSM) is a memory architecture where each processor has its own local memory and the combination of these local memories form a virtual shared memory such that access to shared data is transparent. Mether and Munin are distributed shared memory systems.

- The shared memory of Mether [Fleisch et al (1994), Minnick and Farber (1989), and Minnick (1993)] is implemented by moving memory pages throughout a network based on demand paging concepts, these pages are never moved to disk. However, ensuring consistent memory is the responsibility of the designer/programmer.

- Munin is another DSM system [Fleisch et al (1994), and Carter et al (1991)]. It is based on a single virtual address space that is distributed across machines and memory modules [Bennett et al (1990), and Carter et al (1991)].

- Emerald and Orca are languages supported by DSM systems. Emerald is an object based programming language (similar to Java), and can be classified as a distributed shared memory language [Bal et al (1992)]. Orca's shared data objects are implemented using DSM, and distributed using replication and migration techniques.

**Computational Elements.** The structural nature of computational elements (nodes) of a distributed application varies significantly. For example, processes of client/server applications, virtual processes from the mpC language [Karanov et al (2001), and Arapov et al (1997)], or functions of Mentat programs [Grimshaw (1993)]. Some computational elements are pre-emptive, such as UNIX processes, yet others are atomic such as the execution of code of a Charm object [Kale et al (1994a), and Kale et al (1994b)]. Statements of a computational element might be limited as in CODE [Brown et al (1994)].
The diversity of such elements is clearly illustrated within the literature. Browne et al (1994) state that a computational element is somewhat analogous to a simple statement of a conventional programming language. However, these elements are described elsewhere as macro-level units of computation [Newton et al (1992)]; still further into this paper, the computation element is presented as a function call statement. Furthermore, a CODE program consists of a set of graph instances created at runtime [Newton et al (1992)] which are like subroutines from a common programming language.

**Connectivity.** Connectivity of a distributed application refers to the organisation of computational elements that constitute the application. Connectivity may reflect data flow, control flow, or temporal relationships between tasks of a process. For example:

- The Mentat Programming Language (MPL) is an MIMD object-oriented parallel programming language based on C++ [Grimshaw (1993)]. Parallelism of Mentat programs is transparent because the Mentat compiler detects parallelism to generate data flow graphs such that computational elements are functions and arcs are data flows.

- ALPS (A Language for Process Specification) is a control flow language for specifying a directed graph of precedence and temporal relationships between tasks of a process [Ray and Catron (1991), and Ray (1992)]. That is, some tasks must occur in some predefined order, and some execute in parallel.

Many languages and systems do not support explicit declarations for organising computational elements, i.e., the structure of an application is (relatively) unknown. In some cases, the number of computational elements is actually unknown because of automatic partitioning of the solution into an unknown number of computation elements. In other cases, an invocation of a send function indicates connectivity between at least two computational elements at the time of the invocation. The connectivity of systems based on shared memory, message passing, graphical declarations, parallel processing is presented in the following points.

- For shared memory and distributed shared memory systems the connectivity is based on statements reading from or writing to the memory or accessing shared objects via the object's operations. Such languages are Mether, Munin, Amber, Argus, P4, Charn, Charn++ and Linda

- Some systems support the dynamic creation of computational elements and connectivity is based on function calls, such as send and receive, or data dependencies within the source code. Such languages are PVM, MPI, P4, ABCL, Concurrent Smalltalk, Arjuna, Emerald, Erlang, sC++, SR, Occam,
Mozart, OZ and Distributed OZ. The ORCA language allows dynamic process creation using the fork function and also creation of shared data objects. A shared object can be a parameter of fork causing the object to be passed to the new process and so establishing connectivity.

- Connectivity is naturally explicitly declared when using systems that graphically depict computational elements and communication channels. Such languages are VPE, CODE and HeNCE.

- Applications written in parallel programming languages such as NESL, mpC and pC++ automatically and dynamically create many like computational elements to execute in parallel and perform on a collection of data such as a vector. For example, the nth computational element operates on the nth data element of the vector, and so the overall operation on the vector is executed in parallel.

**Multicomputers.** A multicomputer is a computer consisting of several computational elements, such as processors [Quinn and Hatcher (1990)] or complete computers [Wilkinson et al (1999)], that are connected by a network [Skillicorn et al (1995)]. Shared memory is not involved.

**Multiprocessors.** A multiprocessor is a computer consisting of many independent computational elements [Lafferty et al (1993)], such as processors or computers, connected by shared memory [Skillicorn et al (1995)] or a network [Hudak (1986)]. All processors of a multiprocessor using shared memory may communicate with each other using the shared memory [Wilkinson et al (1999)]. On the other hand, all processors of a multiprocessor using distributed memory communicate via message passing [Hudak (1986)]. A multiprocessor is capable of executing several different programs using shared memory [Krishnamurthy (1989)].

**Parallel Computers.** A parallel computer is a computer that consists of several computing elements, such as processors or complete computers, that are interconnected using shared memory or a network [Skillicorn and Talia (1995), and Wilkinson and Allen (1999)].

**Parallel Programs.** A parallel program in execution on a multicomputer consists of several processes running on several processors in parallel [Andrews et al (1993), and Lafferty et al (1993)]. The processes communicate via message passing only [Skillicorn et al (1995)]; shared memory is not associated with multicomputers. These processes cooperate in order to perform computations to solve a specific problem [Wilkinson et al (1999)]. However, not all parallel programs are restricted to execution on multicomputers. The processes of parallel programs that execute on
shared memory multiprocessors can communicate using available shared memory [Andrews et al (1993)]. However, from [Ben-Ari (1990)], a concurrent program only requires one processor to execute several processes. Such a definition does not support actual parallelism. Processes running under this definition share the CPU.

Several parallel programming languages have been introduced, e.g., ABCL, Charm, Charm++, Orca, pC++ and Mentat. Other parallel programming languages and systems are sC++, NESL, mpC and Express.

- sC++ is an object oriented parallel programming language [Petitpierre (1998)]. However, an object’s operations cannot be used while the object is engaged with synchronous activities. That is, the object’s operations cannot be invoked in parallel by other objects until the activity has terminated.

- NESL is a vector based, purely functional, data-parallel language. Parallelism is achieved by applying operations on vectors, i.e., an operation is applied in parallel to each vector element. NESL is for massively parallel machines (such as CM-2, CM-5, or Cray Y-MP) using shared or distributed memory [Blelloch (1992), Blelloch et al (1994), Blelloch (1995), and Blelloch and Greiner (1996)].

- mpC is a parallel programming language for distributed memory machines, and is supported by MPI [Karganov et al (2001), and Arapov et al (1997)]. However, the mpC programmer cannot determine the number of processes that constitute the parallel application, and cannot determine which computers execute such processes [Lastovetsky (2000)]. An mpC program is executed in a virtual computer space consisting of virtual processors. Virtual networks can exist within the space, and each virtual processor is implemented as a process. Parallelism in mpC is based on three execution modes of function: a function can be executed in parallel by all processes of a program on the entire computer space; a function can be executed in parallel by all processes of the program on a particular virtual network; and a function can be executed in parallel on a virtual processor.

- A system called Express generates parallel programs from sequential programs. This is achieved by applying, not only Express tools, but also a particular development life-cycle on sequential programs. After applying VTOOL and FTOOL tools to a sequential program, the ASPAR tool is used to automatically produce a parallel program [Geist et al (1994)].

---

1. The CM-2 computer can consist of 65536 processors using distributed memory.
2. The CM-5 is limited to 16384 processors using distributed memory.
3. The Cray Y-MP is a shared memory, vector multiprocessor.
Coarse-grain and Fine-grain Parallelism. Coarse-grain and fine-grain parallelism within a parallel program occur when the duration of computation is relatively large or small respectively when compared to the duration of communications [Lafferty et al (1993)]. Distributed applications for this research are coarse-grained.

Loosely and Tightly Coupled Distributed Systems. A distributed system is loosely coupled when the computational elements, i.e., complete computers, are connected by a network, and therefore communications is performed using message passing [Philips (1995), and Krishnamurthy (1989)]. A distributed system is tightly coupled when the computational elements, i.e., a set of processors, are connected to and use shared memory [Krishnamurthy (1989)]. Distributed applications for this research are loosely coupled.

Message Passing. Asynchronous message passing is an extremely efficient form of communication within distributed networks, the sender is not blocked while operations are performed on the message by the receiver [Agla et al (1993), and Ben-Ari (1990)]. In addition, message passing has the following advantages:

- portability [Dongarra et al (1995a)];
- scalability with respect to the number of processors [Dongarra et al (1995a), and Bennett et al (1990)];
- it will be continually available irrespective of the increases in network speeds or computer architectures consisting of shared and distributed memory [Dongarra et al (1995a)];
- increases in the speed of not only processors compared to memory but also networks support the continued usage of message passing [Bennett et al (1990)]; and
- the ability to perform coarse-grained computations on several high performance computers on a network [Lafferty et al (1993)].

Message passing is available in many forms such as asynchronous, synchronous and RPC. Some languages and systems are restricted to one particular form, whereas others incorporate a combination of message passing techniques. The SR language actually includes two forms of asynchronous message passing. The following languages and systems provide a variety of message passing techniques.

- Synchronous message passing is provided by Arjuna and Emerald. Arjuna is a C++ object-oriented programming system for developing distributed applications. It provides RPC for communication between objects, and the message sender blocks for the RPC's return value [Shrivastava et al (1991), and Parrington et al (1995)]. Emerald is an object based programming language and
uses synchronous procedure calls that block the sending object [Raj et al (1991), Hutchinson et al (1991), and Storner et al (1990)].

- Asynchronous message passing is available in several languages such as Actor [Agha et al (1993)], Concurrent Smalltalk [Hopkins et al (1989)], Erlang [Armstrong (1996) and (1997)], and Linda [Ahuja et al (1986)]. Instead of the sender blocking, as in synchronous systems, the sender continues execution. In Erlang programs, the receive operation blocks for a named message, however there is no guarantee that the message will arrive, causing the receiver to block indefinitely.

- Some languages and systems provide combinations of synchronous, asynchronous, and RPC techniques for message passing, for example, PVM [Beguelin et al (1993)], MPI [Dongarra et al (1995a)] and SR [Olsson et al (1992) and Andrews et al (1993)]. The PVM and MPI systems provide both synchronous and asynchronous techniques, but the SR language provides all three.

PVM requires both task and message identification numbers to be sent along with a message. However there is a problem, [Geist et al (1995)], in that these two identification numbers are not sufficient to guarantee that the correct message is received. This is especially the case when messages are sent from library functions, and in such cases the message identification numbers have no meaning outside of these functions.

MPI and PVM require explicit coding to pack data of several types into a buffer prior to sending the message. Similarly, VPE [Newton et al (1995)] requires three functions to send a message: one creates a buffer, one packs the buffer, and another sends the message.

In SR, two asynchronous send operations are provided, not just one. One creates a new process to execute the operation concurrently. The second forces the message into a message queue that is accessible by any other SR process.

Buffers. Two buffers are used to send data within a message passing system. The data in one buffer is sent by the sender, and this data is placed in a second buffer that is accessible to the receiver. The PVM system uses buffers in this way. However, other systems use abstractions that are transparently supported by such buffers to provide message passing. For example:

- Linda [Ahuja et al (1986) and Bal et al (1992)] is based on a tuple-space which is one buffer and global to all processes. A process asynchronously adds a tuple into the space, and then continues execution [Carriero and Gelernter (1989)]. This addition into the space is not directed to any another process. However,
some other process may remove\textsuperscript{1} this tuple, process the data within the tuple, and then add another tuple into the space.

- The SR language is based on several messages queues (buffers) that are global to all processes. In fact, SR message queues are defined outside of SR process specifications. A receiving process must invoke a blocking receive which contains not only the name of the message, but also the same number and type of variables used in the sent message.

- Another arrangement of buffers has them inside processes. PVM, MPI and VPE do this. These systems require the programmer to create a buffer, place data into the buffer, and invoke the send procedure. The receive procedure extracts the data from a second buffer.

- The P4 system relies on buffers for synchronous message passing. However, a significant amount of buffer creation and management must be coded by the programmer [Geist et al (1994)].

**High-Level Languages.** High level languages that are designed to suit particular problems decrease the time to design and implement a distributed application, and also simplify parallel programming [Skillicorn et al (1995)]. Given these advantages, distributed applications for this research are specified using two new high level languages, DAL and DAPI. Both languages and compilers have been designed and developed as part of this project.

### 2.2 Distributed Programming Languages and Systems

**Designing a Distributed Language.** Process spawning, cooperation and termination are the three major issues in designing a language for distributed programming [Skillicorn et al (1995)].

1. **Process Spawning** is the creation of one or more processes during run-time. In the simple case, all processes of an application are created at the time of application instantiation, in which case advantage can be taken during compilation of strong checking of connectivity [Skillicorn et al (1995)]. The creation of processes within the DAS system follows this simple case because an application within the DAS system is viewed as a single entity, and therefore spawning is focused on application creation, i.e., the creation of a set of processes connected in a specified fashion, not just one process.

\textsuperscript{1} A Linda process may remove a tuple when it matches the tuple's format.
Although processes in a DAS application are currently created or spawned statically, it is perceived that future designs of DAS applications will provide for dynamic creation or spawning of DAS applications. Spawning a DAS application, a set of interconnected processes, generalises the specific case of spawning a single process, and therefore application spawning is much preferred to process spawning.

2. **Process cooperation** consists of three main categories [Skillicorn et al (1995)]: explicit message passing, rendezvous communication, and remote procedure call. Message passing can be synchronous, asynchronous, point to point, and one to many [Skillicorn et al (1995)]. Rendezvous communication is synchronous, in which both the receiver and sender block [Skillicorn et al (1995)]. Remote procedure call (RPC) is also synchronous communication [Corbin (1991), and Feit (1993)], a sending process blocks after invoking a remote procedure, and waits for the results of the procedure to be returned by the receiving process.

The selection of asynchronous message passing as the type of process cooperation mechanism within DAS applications is based on efficiency. Message passing languages allow a programmer to produce efficient parallel programs [Arapov et al (1997)]. Message passing within DAS applications is asynchronous, and point to point.

3. **Distributed Termination of Processes** is the termination of all processes of a distributed application. In [Skillicorn et al (1995)] the authors observe 'Notice that in a distributed system, processes communicate by exchanging messages, and there is no central controller to observe the state of all the processes and decide on global termination.' The lack of a central controller implies that management of many distributed applications is quite limited and an operator can not determine the state of a runaway application to appropriately control it. The DAS system is a central controller and can terminate all processes of a DAS application. In addition, a DAS application can be terminated from within by a process requesting termination by the DAS system.

**Developing a Distributed Application.** Developing distributed applications with various control and connectivity characteristics is possible because many different languages exist. In addition, the methodology used to design such applications is dependent on a programming paradigm such as object based or object oriented, data flow or control flow, functional, or process oriented. These applications may also depend on systems such as shared memory, distributed memory, or DSM. Many of these languages are mentioned above when discussing the characteristics of distributed applications. However, some languages, libraries and systems to develop
distributed applications are not covered. Languages for these other paradigms and libraries and systems to develop such distributed applications are presented in this section.

2.2.1 Object Oriented Parallel Programming Languages

The first object oriented language, Simula67, was developed 35 years ago [Dahl and Nygaard (1966)] followed by Smalltalk in 1969 [Ingalls (1978), Goldberg and Robson (1983), and Goldberg (1984)], and CLU in 1974 [Liskov (1992)]. Object-based and object-oriented languages such as Actor, Arjuna, Charm, Charm++, ABCL, Amber, Emerald, pC++ and sC++ are presented above when discussing message passing, distributed memory, shared memory, distributed shared memory, and parallel programs. It is clear that many object oriented languages have been produced for a diverse range of systems.

Three other object oriented parallel programming languages (Argus, Concurrent Smalltalk and Orca) are presented in this section. Argus is based on abstract objects called guardians; concurrent Smalltalk runs on a virtual machine; and Orca (mentioned in discussions on Shared Memory and Distributed Shared Memory in Section 2.1) is object based.

**Argus.** Argus is a programming language for developing distributed programs that use transactions based on guardians and actions [Ghemawat (1990), and Philips (1995)]. Argus programs are expected to execute for long periods of time, such as in banking systems, and to ensure that the program data remains consistent in spite of failures and concurrent access.

A guardian is an abstract object of resources and operations. An RPC is used to invoke an operation by sending the operation name and arguments to a guardian. These are sent using a ‘call message’, and results are returned using a ‘return message’. In addition, on receiving a call message, a new process is created to process the message. The state of a guardian is periodically saved to a file system to support recovery from failures.

**Concurrent Smalltalk.** Concurrent Smalltalk [Hopkins and Woleczko (1989)] is an object oriented concurrent programming language. Concurrent Smalltalk programs are supported by the traditional Smalltalk virtual machine and asynchronous message passing. However, processes are bound to this virtual machine, and therefore are not distributed over a network. Concurrent Smalltalk is designed for fine grained processing [Kale et al (1993)].
Orca. Orca is an object based parallel programming language for loosely coupled distributed applications [Bal (1990), Bal et al (1992), Bal et al (1998), and Bal et al (1990)]. An Orca application starts with one process only and uses fork to create additional processes. Therefore, the connectivity of Orca processes at startup is application dependent and although controlled is unknown.

2.2.2 Control Flow and Data Flow Languages

Control flow languages are characterised by language constructs, such as selection and repetition, that temporally control which computational elements are to be executed. Examples of control flow languages are ALPS (presented in section 2.1 within discussions on connectivity of computational elements) and HeNCE discussed below.

Data flow languages are characterised by a specific model of computation. In this model a computational element that depends on, say, N data dependencies does not commence execution until all N dependencies are available. For example, a function of three variables, say f(x, y, z), is evaluated when the three variables are assigned values. Examples of data flow languages are Mentat (presented in section 2.1 within discussions on connectivity), CODE, Oz, Distributed OZ and Mozart which are discussed below.

HeNCE. HeNCE is a parallel programming environment for specifying a parallel application using a graph [Beguelin et al (1993), and Beguelin et al (1994)]. Nodes represent procedures such as functions in C. Arcs represent data dependencies or control flow. With respect to data dependency, a node at the tail of an arc must complete execution before the node at the head of the arc commences execution. In addition, a node commences execution when all of its preceding nodes have completed their execution. Data is sent to a node from all preceding nodes. Arcs representing control flow symbolise programming constructs such as repetition and selection. The HeNCE system takes such a graph to generate source code which is destined to execute on the PVM system [Beguelin et al (1993)].

CODE. CODE is very similar in purpose and philosophy to HeNCE [Browne et al (1994)]. In CODE, a graph is used to specify a parallel program, and computational nodes of both CODE and HeNCE are procedures. CODE graphs depict data flow and HeNCE graphs depict data dependencies or control flow [Browne et al (1994)]. Both CODE and HeNCE have serious problems [Newton et al (1995)] because the graphs are complicated, awkward and large.
Chapter 2. Literature Review

Computation commences when firing rules of nodes are satisfied [Browne et al. (1994)]. The author's state that these firing rules are complicated and wordy, and that CODE programmers must precisely specify the conditions under which a node commences execution. Computational nodes within CODE and HeNCE only communicate at the beginning and end of a computation [Newton et al. (1995)].

**Oz.** Oz is a thread based, shared memory, data-flow, object-oriented programming language [Haridi et al. (1998), Popov (1997), Lafferty et al. (1993), and Ackerman (1982)] for producing functional, logical, constraint and object oriented programs. It can be used to produce such programs instead of languages such as Lisp, Prolog and Smalltalk [Smolka (1995)]. Computational elements are fine-grained data-flow threads that execute sequential code. A thread blocks until all of its data dependencies are available [Haridi et al. (1998)]. An early version of Oz is object based [Popov (1997)], but a later version is object oriented [Haridi (1997)]. Oz variables are unique in that they are single-assignment variables. That is, only one value can ever be assigned to an Oz variable.

Data flow graph nodes are said to fire (start execution) when there is data at each of the node's incoming arcs. Such data flow firing rules match those of Petri nets [Peterson (1981), Dew (1992)]. In brief, a Petri net is a directed bipartite graph that can be used to model and mathematically analyze parallel systems. Several nodes can simultaneously fire. After the input data are absorbed by a node, and the function completes execution, then the results are placed on all outgoing arcs to be sent to other nodes. The literature on Petri nets is substantial. Introductory references are [Peterson (1981), Agerwala (1979), Muarta (1989), Sifakis (1980), Zuberek (1980), Zuberek (1987), and Marsan (1989)].

**Distributed Oz.** Distributed Oz extends Oz by supporting two additional concepts [Haridi et al. (1997)]: mobility control of objects and asynchronous ordered communication. The network is hidden from the programmer, it is the Oz system that invokes network operations. Message passing operations such as send and receive are not available to the programmer.

**Mozart.** Mozart is a programming system that uses Oz for producing thread based, constraint based, symbolic, logic, distributed data-flow programs [Van Roy and Haridi (1999), and Van Roy (1999)]. Although processes are distributed over a network, the network is actually hidden from the programmer and therefore the specification of process location is not possible.
2.2.3 Functional Languages

Programming languages classed as purely functional are those such that all expressions are specified using functions. Therefore, evaluation of an expression is essentially the evaluation of a function. For example, an expression to sum four variables can be \( x = a + b + c + d \). However, this expression in a functional language would be based on a function such as \( x = \text{sum}(a, b, c, d) \).

NESL and Erlang are parallel programming languages. NESL (mentioned in discussions on Parallel Programs in Section 2.1) is a purely functional parallel language. Erlang evolved into a functional language from a logical one like Prolog.

**Erlang.** Erlang is a functional, message passing, parallel programming language for soft real-time systems [Armstrong (1996) and (1997)]. The Erlang compiler generates virtual machine code, and therefore Erlang programs execute on a virtual machine.

An Erlang process can only send a message to another Erlang process if the sender has the process identification number of the receiver. Therefore, an Erlang programmer must write code to send process id values from process to process in order to establish connectivity between two processes. In addition, the connectivity of an Erlang application is not explicit within an Erlang program, i.e., all processes of an Erlang application must be created one at a time during runtime. An Erlang application can only be distributed over several computers if each computer is running an Erlang daemon.

2.2.4 Process Oriented Languages

The computational element of process oriented languages is a process. Such languages are mpC, Occam, SR and VPE. mpC (mentioned in discussions on Computational Elements and Parallel Programs in Section 2.1 Characteristics of Distributed Programming) is based on a virtual computer space, and a function is executed\(^1\) in parallel on virtual processors. The Occam language is process oriented and is based on synchronous message passing. SR (mentioned in discussions on Message Passing and Buffers in Section 2.1) is based on synchronous resources, asynchronous message passing, and messages are sent to buffers.

**Occam.** Occam is a synchronous message passing concurrent language [Skillicorn et al (1995)]. It is based on CSP (Communicating Sequential Processes [Hoare (1978) and (1985)]) and so formal proofs of program correctness can be undertaken. Occam was specifically designed for transputers [Skillicorn et al (1995), and May et al (1987)], but it lacks dynamic memory allocation [Skillicorn et al (1995)].

---

\(^1\) Execution of mpC functions is similar to that of parallel operations in NESL.
SR. SR (Synchronizing Resources) is a concurrent language that incorporates local and remote procedure calls, rendezvous, message passing, dynamic process creation, multicast, and semaphores [Olsson et al. (1992), Andrews et al. (1993), and Gebala and McNamee (1995)]. The main language constructs are resources, globals and operations:

- Resources are templates for creating instances of executable code. They encapsulate declarations, statements, processes, and variables shared by processes. In addition, instances can be allocated to specified processors.
- A global is a resource instance without parameters.
- Operations are functions used by processes for communication. They may be synchronous or asynchronous when using a call or send statement respectively.

Messages of an SR application are sent to message queues (not to any process) and the connectivity of processes is implied by call, send and receive statements embedded within the source code of an SR program.

VPE, VPE (Visual Programming Environment) is a process oriented, message passing, parallel programming environment [Newton et al. (1995)]. The authors are from the groups that produced CODE and HeNCE. Nodes of a VPE graph represent processes, and arcs represent communication channels on which messages are transferred. An arc is connected to two processes via named ports, one port at each end of the arc.

A message is sent from one process to another using three VPE library functions: a buffer is created by the first function, the message is stored into the buffer by the second function, and the message is sent using the third function.

A message is obtained by the receiving process using two functions. The first function blocks the receiving process on the named port. The second function is used to extracts the message from the buffer, and then places the message at some specified memory address.

In VPE, a communication channel starts and ends at named ports. These port names are required for both sending and receiving messages. Newton et al. (1995) present a non-blocking receive function in VPE, however they do not discuss its semantics. In addition, VPE messages are not typed, and may consist of an unknown arrangement of several types [Newton et al. (1995)].

### 2.2.5 Shared Memory Systems

Two methods are available for communication among processes: shared memory and message passing. Some systems allow chunks of memory to be shared, while others share memory using abstract data structures, or are based on distributed shared
memory that is implemented using message passing. For example, Ivy is based on network-wide virtual shared memory and Linda is based on a tuple space that supports distributed data structures (Linda is mentioned in discussions on Buffers in Section 2.1). Mether and Munin use distributed shared memory (both are mentioned in discussions on Distributed Shared Memory in Section 2.1).

Ivy. Ivy [Li (1988)] is a loosely coupled multiprocessor system using a network-wide virtual shared memory [Chase et al (1989), and Fleisch et al (1994)]. Shared data is paged between processors [Fleisch et al (1994)], and data on these replicated pages is consistent at the byte level [Chase et al (1989)].

2.2.6 Library and Tool Systems

Distributed programs are essentially based on message passing, including those that use distributed shared memory because this is implemented using message passing. One technique to develop a distributed program is to use communications libraries that are part of, or augment, the host operating system. For example, a socket library provides send and receive functions for remote communications. However, distributed programs are tedious to write using such UNIX libraries because these libraries provide low level functionality.

Several libraries and tools have been developed to provide high level functionality with respect to distributed programming. For example, objects and object oriented message passing is popular. Other systems such as CORBA, MPI, P4, PVM, Mozart and Express provide send and receive functions, and abstract away the requirement to manage sockets. Mozart is mentioned in Section 2.2.2, and Express in Parallel Programs of Section 2.1.

CORBA. CORBA is an object oriented system for developing distributed applications. In 1991 the Object Management Group (OMG) published revision 1.1 of the Common Object Request Broker Architecture (CORBA) specification [Vinoski (1993)]. This is a specification of essential interfaces and services of a mechanism that transparently provides object registration, object location, object activation, and client-object communication [Vinoski (1997)].

The Object Request Broker (ORB) not only delivers messages from clients to objects, but also returns results to the clients (from the objects) [Vinoski (1997)]. In addition, it hides the location of the target object, how it is implemented, whether it is ready to accept requests, and what communications mechanisms (TCP, UDP, shared memory, etc.) are used to deliver messages and return results.
MPI. MPI is a communication interface layer based on message passing [Geist et al (1994)]. It provides send and receive operations for passing typed messages among concurrent processes executing on heterogeneous machines [Dongarra et al (1995a)]. However, it is not a software system for producing complete distributed programs [Dongarra et al (1996)]. Furthermore, MPI send and receive functions require explicit programming to pack and unpack data that are of different types [Dongarra et al (1995b)].

P4. The P4 system is a library for shared memory, synchronous message passing, parallel programming [Geist et al (1994), Butler and Lusk (1992a), and Butler and Lusk (1992b)]. Monitors are used when programming for shared memory.

Although messages are typed, the type is not predetermined, such as int, float or char as in C. A message type is just an integer value that is provided by the programmer, and therefore, the meaning of such types is completely determined by the programmer. In addition, the length of a message must be a parameter of the send operation representing the number of bytes within the message.

A distributed application based on P4 requires a startup configuration file in order to specify hosts machines, file names of object modules, and the number of processes to create per machine using these object modules. However, the connectivity of such processes of a P4 application is not explicitly specified, but is implied by the usage of send and receive statements within the program.

PVM. The PVM (Parallel Virtual Machine) system is a library of low level tools [Beguelin et al (1993)] for developing message passing, parallel programs, based on C and Fortran, that can execute on a virtual parallel computer. This virtual computer can be implemented on a network of heterogeneous UNIX computers [Beguelin et al (1993), Geist et al (1994), and Sunderam (1990)], and message passing can be synchronous or asynchronous [Beguelin et al (1995)]. In addition to message passing functions, the PVM library provides functions such as those to start PVM tasks (UNIX processes), pack data into PVM buffers, send such packed data, and receive and unpack data.

The PVM system allows a PVM task to send data to any other PVM task based on a task identification number [Beguelin et al (1995)]. PVM does not provide notation for explicitly specifying the connectivity of an application's processes, but the connectivity is implied by the send and receive function calls within the program.

PVM views processing entities as PVM tasks but does not view several cooperating tasks as a single entity. PVM and MPI provide little assistance in controlling the complexity of parallel programming [Kale et al (1994a)].
2.3 Project Requirements

An introduction to distributed applications using message passing associated with this research is presented in Chapter 1. Distributed applications generally consist of several computational elements distributed over a network that communicate by message passing (at the primitive level) or distributed shared memory (at a higher level of abstraction). Therefore a distributed application of the sort relevant to this project requires computational elements, a mechanism to remotely distribute and manage these elements, a library of high level message passing functions, a medium to transmit messages, and buffers to store messages. This project requires:

- languages to specify applications such that connectivity specifications are independent of functionality specifications, enabling separate specification of an application’s structure from the specification of its functionality;
- a language to specify connectivity in a modular fashion to provide re-use of existing connectivity specifications;
- a system to build applications by combining specifications of functional components with connectivity specifications, and then compiling this combined specification to produce executable object modules;
- a system to distribute object modules on a network in order to execute a distributed application; and
- a system to manage several running distributed applications using one process.

The DAL grammar supports statements to specify connectivity and it completely excludes the possibility of writing statements of functionality. The DAPL grammar allows specification of functionality and it excludes connectivity statements, i.e., it is impossible to derive the structural connectivity of an application by analysing DAPL programs. The DAL and DAPL languages intrinsically separate connectivity from functionality, and so provide a new technology for building distributed applications.

Designing software using modules and re-using modules from the same application or pre-written modules from other applications increases the understandability and manageability of the application. Szyperski (1998) describes several kinds of re-use as follows:

- dogma re-use: established principles associated with particular programming or scripting languages to develop code fragments, procedures, functions, etc.,
- library re-use: reusing pre-written libraries functions such a sin() and cos() from a maths library, or strlen() and strcmp() from a C library,
- interface re-use: reusing interfaces and their contracts such as those associated with clients and servers,
- pattern re-use: a pattern is a small architecture of objects for solving a problem,
Chapter 2. Literature Review

- framework re-use: a framework which is a group of classes for solving a particular problem, e.g., the MVC classes (Model, View and Controller classes) that are especially used within Smalltalk applications, and

- architecture re-use: a common architecture is that of an operating system, it can be described as several layers of functionality where each layer is dependent on the functionality provided by its adjacent lower layer. Other architectures of such layers are presented in Szyperski (1998).

Library re-use is the type of re-use that is used in the design of distributed applications of this project. This re-use is achieved by including \texttt{requires} clauses when writing a DAL specification and means that an existing DAL specification is part of the current DAL specification that is under development.

The synthesis of the above requirements for the development of distributed applications is new work, none similar appearing within the literature. The detailed requirements for developing, instantiating and managing distributed application for this research project are presented in the following sections.

2.3.1 DAS Requirements

The system to manage several distributed applications within this project is called DAS (Distributed Application System). It has a textual user interface called DASH (Distributed Application SHELL) that allows the DAS operator to instantiate and manage several distributed applications using DASH commands.

The DAS system is centralised. Only one process is required to control the entire DAS system. It has an application table containing details about each application such as the name and status of the application, names of the (remote) hosts on which application processes are running, and port numbers\footnote{The DAS system records a host name and port number for each application process to enable it to communicate with them.} associated with each application process. Each application has an identity within DAS, and is not just a group of processes as in other systems such as PVM.

The DASH shell is embedded into DAS as one of its functions. It is required to respond to user commands to:

- load application specifications,
- compile application specifications,
- add host names to the system,
- instantiate an application using local and remote processes,
- terminate an application, and
Chapter 2. Literature Review

- reallocate an application process, i.e., to terminate an application process and create another in its place on a different computer, before its host computer is shut down for, say, maintenance or hardware re-configuration.

As well as responding to DASII commands from the user, DAS must react to messages received from applications. For example, when an application completes its processing, it sends a message to inform DAS of its termination. This allows the DAS system to update the application table, and ensure that all processes of this application have terminated.

2.3.2 Distributed Application Requirements

A distributed application consists of a known number of computational elements distributed over a network of computers. The connectivity of these elements is represented by the communication channels between these elements, and these channels provide the medium for sending messages between processes within an application. A message which arrives at a process is automatically placed in a buffer within this process. The receiving process may remove and process the message, and as a result may send another message to another process.

The following five sections contain requirements for computational elements, communication channels, messages, buffers, and DA (Distributed Application) library functions.

2.3.2.1 Computational Elements

In execution, computational elements are processes such as UNIX processes that are specified using C functions, each element is similar to a C program in that main() and other functions are required. They are coarse grained, and distributed over a network of loosely coupled workstations. Their connectivity and location is explicitly specified using the DAL language, and functionality is specified using DAPL.

These elements are reactive and block, waiting for a message to arrive via any of their incoming communications channels. When a message arrives at an element, it processes that message, and then blocks for another message.

A consequence of separating specifications of connectivity and functionality is that the functionality of computational elements may be replaced easily with new versions without altering the connectivity specification. For example, a distributed application might consist of three processes (p1, p2 and p3) that are connected in some fashion. The functionality of these processes might be specified within three files (say, p1a, p2a and p3a), and new versions might be specified within three other files (say, p1b, p2b and p3b). Therefore, replacing the functionality of p1 is as simple as referring to p1b instead of p1a during compilation.
2.3.2.2 Communication Channels
Message passing within DAS applications is asynchronous and supported by one-way communications channels. This allows the sending process to continue execution without waiting for a response, and so computation throughput is increased. For cases requiring a response, another channel from the receiver to the sender can be included to deliver the response. A channel starts at one computational element, and is logically connected at the other end to a buffer within the destination element. Furthermore, several channels from the same element or other elements can be connected to a particular buffer.

Communication channels are named and typed. They are named by the programmer for referencing channels within program statements. A channel of type T means that variable length arrays of T type elements can be passed along this channel.

2.3.2.3 Messages
Message passing within DAS is asynchronous. A message is an array of elements, all of the same data type. This type must match the type associated with the communication channel in which the message is transferred. Therefore there is no need to include type information within a DAS message.

Sending a message can occur at any time during a computation, and requires three values: the address of an array to be sent, the number of elements, and the name of a communication channel on which the data is to be transferred. The send operation is a DAS library function and is built using TCP so that messages are not lost. There is no requirement for a receive operation because a message is automatically placed into the buffer of the destination computational element identified by the channel.

2.3.2.4 Buffers
A buffer within a DAS application is a FIFO queue that is named and typed. It is local to a process, and a process may contain zero or more buffers. Buffer elements are variable length arrays of the same type as the buffer. Removing a buffer element means taking the head element (an array) out of the FIFO queue, by using a DAS library function, and returning it to the function that invoked the library function.

2.3.2.5 Library Functions
Library functions for asynchronous message passing and buffer management are required to support the distributed application programmer. An asynchronous send function (DA_send) will require only three parameters as mentioned in Section 2.3.2.3. The underlying DAS system automatically transfers and places the data into the buffer that is located at the end of the channel. The receiving process can extract the data from this buffer by using a buffer management library function (DA_remove). This function only requires the buffer name, and returns the data.
Termination of an application requires that all of its processes are terminated. Therefore, a library function (DA_exitapp) to invoke application termination is required. In addition, a library function (DA_exit) is used to terminate individual processes.

2.3.3 DAL and DAPL Programming Language Requirements

The connectivity and the functionality of the computational elements are relatively independent, except for ensuring that these elements can be connected to each other. Therefore, they can be specified relatively independently, and hence, two languages can be used to specify each part. Detailed requirements of these languages are presented in the following two sections.

2.3.3.1 DAL Requirements

The DAL (Distributed Application Language) language is used to specify the following characteristics of a distributed application using a minimum number of statements:

- application identity (a programmer supplied name),
- a set of computers which the application is permitted to use,
- the order in which the application's processes commence execution,
- for each process:
  - process identity (a programmer supplied name);
  - a subset of the above computers, on one of which the process will run;
  - name and type of each buffer;
  - name and type of each communication channel that leaves the process;
  - for connectivity - the name of the destination buffer to which a communication channel is connected, and the name of the process in which this buffer exists; and
  - another application name (optional) - using this option establishes code re-use and a hierarchical specification; effectively the named application (of several processes) replaces this single process.

2.3.3.2 DAPL Requirements

The DAPL (Distributed Application Process Language) language is used to specify the following characteristics of each process:

- process identity (a programmer supplied name);
- structure definitions (like struct in C);
- name and type of each buffer;
- name and type of each communications channel; and
- program statements for functionality, data and message passing.
2.4 Conclusion

The literature contains many languages, libraries and tools to support the development of distributed applications that depend on special characteristics such as synchronous or asynchronous message passing, shared memory, DSM, loosely-coupled or tightly coupled processors, and abstractions like tuple-spaces. Distributed applications also differ in their usage of message passing. For example, some messages require the process ID of the destination process, whereas other applications do not send messages to processes but place messages within a buffer that is accessible by other processes. Buffers may be global to all processes as in SR, or local to processes as in PVM. Linda places messages (tuples) into only one global buffer (the tuple space).

Developing distributed applications may be based on programming paradigms such as message passing, DSM, object based, object oriented, functional, data flow, and control flow. In addition, libraries and tools for such development exist. Libraries like that of PVM can support application development based on such paradigms. Tools can convert sequential programs into parallel programs, but this means the application designer has no control of the organisation of computational elements.

Based on the requirements for developing distributed applications for this project, the literature does not include languages, libraries or tools that satisfies these requirements. Developing distributed applications within this project is based on two new languages (DAL and DAPL) and their compilers, and a new run-time system (DAS) to manage such applications.
3 Distributed Applications: Logical Design

Distributed applications are typically described as communicating processes distributed over a network of computers. The requirements for the sort of distributed application in this research are presented in Section 2.3.2 and requirements for their specification are presented in Section 2.3.3. In summary, these distributed applications require computational elements, communication channels, messages, buffers, and a small set of library functions. Their specification of an application consists of two independent parts: the specification of the connectivity between processes, and the specification of the functionality of the processes. The logical design of such distributed applications is presented in this chapter.

The distributed applications of interest in this research consist of communicating processes distributed over a network of workstations. Applications are managed by the run-time system called DAS created as part of the compilation process. The workstations are loosely coupled, supported by one of more file systems. Figure 3.1 depicts two distributed applications and the DAS process. One application has five processes, the other has three. The directed arcs represent one way communication channels. The shaded rectangles represent workstations, four being used altogether.

![Diagram of distributed applications and DAS process]

**Figure 3.1** Two applications and the DAS process distributed on four workstations.

The depiction of distributed applications in Figure 3.1 shows that applications consist of a number of processes that are connected by communications channels. Such connectivity is an essential characteristic of distributed applications and is independent of the internal workings of processes. Therefore, the design of an application can be partitioned into designs of connectivity and of functionality. The
logical design of application connectivity is presented in Section 3.1 and of application processes in Section 3.2.

### 3.1 Connectivity of Application Processes

The logical design of the connectivity of a distributed application is associated with the number of processes of the application and the communication channels that connect these processes. However, data types, the set of computers being used to run application processes, and the order in which to create processes are also characteristics that may be included in the logical design. In addition, a modular way to design an application is achieved by linking a process of the current design to the connectivity design of an existing application. Therefore the connectivity design of an application can be composed of many existing designs which can be composed of other existing designs, and so on. The effect of such linking is that the compilation system replaces the process with all processes of the existing application.

The logical design of the connectivity part of a distributed application consists of information regarding:

- data types,
- communication channels,
- processes,
- process-buffer pairs,
- interfaces for modular design,
- computer names, and
- a list of process names.

#### 3.1.1 Data Types

Processes of distributed applications used in this research communicate using message passing, and the messages are variable length arrays of a specified type. Therefore types are required as part of the connectivity design to ensure that data sent by one process to another via a communication channel is of the same type as the channel, and also the buffer to which the channel connects.

#### 3.1.2 Communication Channels

The connectivity of an application is defined by the way in which processes are connected by communication channels. A channel is unidirectional and directly connects the source process to a buffer in the destination process. In addition, several channels from the same or other processes may be connected to the same buffer.
3.1.3 Processes
In addition to communication channels being part of a connectivity design, processes are also essential to such designs, ensuring the channels are connected to the appropriate processes.

The design of connectivity is not concerned with functionality, but only with process interface details such as process identity, buffer identity, and channel identity.

3.1.4 Process-Buffer Pair
Both the process and buffer identity is required to identify a buffer to which the destination of a channel connects. Buffer identity alone is insufficient because although buffer names are unique within a process, they are not unique among processes. For example, the name of a buffer in one process can be the same as the name of another buffer in another process. However, because process names are unique within an application, a buffer can be identified uniquely in an application by using the process name as well.

3.1.5 Interfaces for Modular Design
The connectivity design of an application can be developed in a modular way by referring to existing designs. This is based on associating the identity of an existing application with a process of the current connectivity design, and also ensuring that the same number and type of channels that enter and leave the process match those of the identified application. That is, ensuring the interface of the existing application matches that of the process.

The interface of a process (application) is the communication channels that enter and leave that process (application). The interface of an application consists of special communication channels. A channel that enters an application is connected to a buffer of a destination process within that application, but is not connected to a source process. A channel that leaves an application is connected to a source process, but is not connected to a buffer.

3.1.6 Computer Names
Processes of a distributed application run on several computers. Therefore, a set of computer names can be used to constrain the process allocation algorithm to create processes on only those computers (listed) within the set. Although such a set might be sufficient for many processes, other processes may be required to run on specific computers. Therefore, in addition to a set of computer names associated with an application, a subset of computer names may also be associated with a process.
3.1.7 Creating Processes in Sequence

Creating a distributed application means that many processes are created on several computers. However, although an application design implies static creation of many processes, then without some kind of process creation management, it is expected (due to network and computer loads) that these processes will be created and start running in some different order each time the application is created, and this may at times cause incorrect results. Therefore, there is a need to ensure the order in which processes are created is constant when the application is created. This problem can be solved by creating processes based on a specified (ordered) list of process names that is provided by the designer.

3.2 Application Processes

Although the processes of the distributed application are implemented in this research using UNIX [Bach (1990), Stevens (1990) and Kernighan and Pike (1984)], each process has the same logical design independent of UNIX, consisting of:

- common operational semantics, i.e., the way a process commences its execution, sends data, and responds to incoming data,
- a set of buffers,
- a set of outputs, and
- a code section.

3.2.1 Operational Semantics

In general, each process is first initialised, then waits for data to arrive in one of its buffers, and processes it on arrival. The process does not terminate after processing data, but continues its existence by waiting for more data to arrive and processing this data.

When the process is created, an initialising function called main() is invoked. This performs tasks such as initialising variables local to the process. On completion, the process blocks for data to arrive in any of its input buffers.

The buffers of a process are global to all user defined functions of the process. Data that arrives at the process is automatically placed in a buffer associated with the input channel, this association being specified using the DAL language. In the event of data being placed in a buffer, an optional function (if defined) is automatically invoked. This function can remove the data from the buffer, and process it. In order to avoid buffer overflow, it may seem essential to provide such a function linked to data arrival. However, this is not necessary because any function may take data from a buffer. That is, buffer overflow can be avoided as the indirect consequence of other processing.
During execution of a process's function, data may be sent to (one or more) other processes. Placing data into buffers of other processes causes associated functions to be invoked.

The logical view of a process is depicted by the circle in Figure 3.2. Arrows entering the circle represent input communication channels. Several communication channels labelled \(i_1\) to \(i_5\) from other processes are associated with this process. The arrows labelled \(o_1\) to \(o_6\) leaving the circle represent output communication channels. Four buffers are depicted as squares. Six functions depicted as ellipses and labelled \(f_1\) to \(f_5\) and \text{main()}\) are associated with this process.

![Figure 3.2 The logical view of a process in a distributed application.](image)

In addition, two types of relationships depicted by straight lines are shown. One type of relationship links input channels to buffers. Data being sent to the process via the input is placed into the linked buffer. For example, data arriving from another process via inputs \(i_3\) or \(i_4\) are placed into buffer \(b_3\). The second relationship is between buffers and functions which indicates a function is invoked when data is placed into the buffer. For example, function \(f_3\) is invoked when data from either input \(i_3\) or \(i_4\) is placed into buffer \(b_3\). There is no need to specify relationships between functions and output channels because all output channels are in the scope of the functions.
3.2.2 Buffers

A process has a set of buffers, possibly an empty set. If there are no buffers, the process cannot receive data. However, it is still possible for the process to send data elsewhere by using the initialisation function called main(), or any other function subsequently invoked from main(), to send the data.

A buffer is a first-in first-out queue of elements, each being a variable length array of some type. This allows an array of data to be sent from one process to another by using one procedure call. Sending an array of data is more efficient than sending each item one at a time. Especially with the transmission of text strings, it is more efficient to send a string (array) of characters to another process rather than sending each individual character. In general, one is able to transmit an array of any specified type (char, int, struct, etc.) of any length to another process.

Three actions occur as a result of data arriving at a process. The first is that the data is placed in the buffer that is linked to the input channel on which the data arrived. For example, in Figure 3.2, data sent via the input i₄ is placed in buffer b₃. The second action is that the associated function, if any, of the buffer is invoked. For example, in Figure 3.2, function f₃ is invoked when data is placed in buffer b₃. The third action is that the process blocks, waiting for more data, when the function returns.

If a buffer has no linked function, then the second action cannot occur. More catastrophic is that buffer overflow may occur when the process has an empty set of functions because no function exists to remove data from the buffer. However, processes without functions are impractical and very unlikely to exist.

When functions exist within a process and none are linked to a buffer, then this does not (necessarily) mean that buffer overflow will occur. It is possible that another function could remove data from this buffer using the library function called DA_remove(). For example, in Figure 3.2, buffer b₄ is not linked to a function and therefore buffer overflow may eventuate. However, when data is placed in buffer b₁, function f₁ is invoked and can consequently invoke DA_remove(b₄) to remove data from b₄.

In general, any function can invoke DA_remove(), and as buffers are global to all functions within a process, then data can be removed from any buffer by any

---

1. Library functions of the distributed application system that are available to the programmer are prefixed with DA (distributed application).
function that invokes DA_remove(). This library function has one parameter which is the name of a buffer, and returns the element removed from the buffer. For example,

\[
data = DA\_remove(buffer\_name);
\]

### 3.2.3 Outputs
A process sends data to another process using a communication channel which is connected to both processes. Each channel is typed and linked to a buffer of the destination process. Data sent via this channel is placed in the buffer. In addition, many channels can be linked to the same buffer. For example, in Figure 3.2, two channels named \(i_3\) and \(i_4\) are linked to buffer \(b_3\).

Although a process can be specified with a set of output communication channels, it can also be specified with no outputs. In that case, the process cannot send data to other processes. However, such a process is still useful, for example, to write information about incoming data to one or more files, or to react to incoming data by operating hardware devices such as a printer, or some medical actuator.

An output is used to send data from one process to another. Logically, the sending process is aware of the output channel, but is unaware of the receiving process, and it places data into this output channel to transfer the data. The underlying system takes care of the physical transmission of bytes from the sending process to the buffer of the destination process.

The format of the data which is placed into an output channel is a variable length array of a specified type. The type is also the type of the output channel and the buffer on the other end of the output channel. For example, if the type of an output channel is character, then a string (array) of characters can be transferred from one process to another.

The library function used to transmit data is called DA_send(). This function requires three parameters: the name of an output channel, the location of an array of data, and the number of elements to be placed into the output. The value returned from DA_send() indicates the number of bytes transmitted, or an error value. The signature is,

\[
n = DA\_send(output\_name, location, count)
\]

### 3.2.4 Functions
A process has a set of functions, which can be empty. An initialisation function called main() can be declared, and if so, it starts immediately after the instantiation of the logical process. This initialisation function can be used to either initialise, say, process
variables or commence any other necessary start up activities. Following the
execution of the initialisation function, the process blocks for data to arrive on one of
its input channels.

If the set of functions of a process is empty, then the process has no
functionality, cannot remove data from buffers, and is practically useless. Although
the set of functions may be empty in principle, it is never so in practice.

3.3 Summary
The logical view of the processes of a particular class of distributed application is
described in the above sections. Such a distributed application is reactive. That is, the
processes of an application block, waiting for data to be placed into one of their
buffers. On arrival of data, a process function may be invoked to process the data.
Processing the data may result in new data being sent to one or more other processes,
causing further reaction, and so on.

The external components of such distributed applications are processes and
communication channels supporting the transmission of logical data packets
containing variable length arrays of data elements. Internally, each process is
composed of buffers, functions and outputs. The type of data sent from one process to
another must be the same type as the output, communication channel, and destination
buffer.

By separating the communication between processes from the functionality of
each process, the structural connection of the distributed application may be specified
separately from the functionality. Such specifications can be written using the DAL
and DAPI. languages which are presented in the following two chapters (Chapter 4
and Chapter 5).
4 Specifying Distributed Applications Using DAL and DAPL

Two new programming languages, called DAL and DAPL, have been created for specifying distributed applications are described in this chapter. They allow a programmer to specify essential characteristics of these applications such as: where processes are, what processes do, data structures of processes, and the connectivity of processes. DAS library functions provide primitives such as `DA_send()` for DAL and DAPL. In addition, a graphical notation, similar to data flow diagrams, is presented in this chapter. This graphical notation provides a symbolic view of an application’s DAL specification.

The DAL and DAPL languages allow a programmer to specify the following characteristics of a distributed application:

- Process functionality and data structures.
- Connectivity between processes via interprocess communication channels and buffers.
- A general set of host machine names to which processes can be allocated.
- A set of host machine names to which a particular process can be allocated. A process may need to be allocated to a particular host because of unique resources available there. This set is a subset of the above set, and if empty the DAS system resorts to the general set of host names.
- A list of process names that specifies the order in which processes are created and started.

In the following three sections a distributed application is specified using a graphical notation, the DAL language, and the DAPL language. DAS library user functions are presented in Section 4.4. This is followed by an example distributed application in Section 4.5, and conclusions in Section 4.6. Additional statements for hierarchical specifications are presented in Chapter 5.

4.1 Distributed Application Diagram

The graphical notation called the distributed application diagram is used to depict the application’s processes and connectivity. It graphs a subset of information that can be specified using the DAL language, and quickly provides a clear picture of the connectivity, whereas the DAL language provides only textual specification. Although the connectivity is simple to specify in DAL it can be quite difficult to imagine, so defining a graphical notation and a mapping to DAL is important to the user. Labelled symbols are used to represent processes, outputs and buffers.
The distributed application diagram does not include specifications of process details such as functionality and data structures. These process specifications are produced separately and written in the DAPL language. The graphical notation is shown in Figure 4.1.

![Diagram](image)

**Figure 4.1** Graphical components of a distributed application diagram.

The diagram and components are labelled. For example:
- The diagram is labelled with the name of the distributed application, e.g.,
  ```
  example_1
  ```
- Each process is named, e.g.,
  ```
  proc_1
  ```
- Each output is typed and named, e.g.,
  ```
  int a
  ```
- Each buffer is typed and named, e.g.,
  ```
  int buf_1
  ```

A process is depicted in a diagram of a distributed application using an ellipse and a label which is the name of the process. For example, there are two processes named `proc_1` and `proc_2` in the application `example_1` as shown in Figure 4.2.

![Diagram](image)

**Figure 4.2** A diagram called `example_1` of two processes.

A buffer is depicted in a process symbol as a square, labelled with its name and type. For example, in Figure 4.3 `proc_1` has a buffer called `buf_1` for integer data. The buffer in the second process is similarly labelled. Buffer names in the same process must be unique.
A process communicates with other processes by sending data via a uni-directional communication channel, known as an output. An output exits one process (the sending process) and enters another (the receiving process). Each output is drawn as a directed arc labelled with the type of data it carries and its name. For example, two outputs appear in the diagram depicted in Figure 4.4. One output of integer type called \( b \) carries data from process \( \text{proc}_1 \) to \( \text{proc}_2 \). Similarly, the second output is named \( a \) and carries integer data from \( \text{proc}_2 \) to \( \text{proc}_1 \). As for a buffer, output names in a process must be unique.

Figure 4.4 also shows that each output is associated with a buffer of the destination process. Data sent via an output is placed into the associated buffer. This association is depicted by a short line that is drawn from the arrow-head of the directed arc to the square representing the buffer. So an integer sent from process \( \text{proc}_1 \) via output \( a \) is placed into buffer \( \text{buf}_2 \) of \( \text{proc}_2 \).

In addition, a many-to-one relationship exists between outputs and buffers. That is, many outputs can be associated with the same buffer, so data from several processes can arrive in the same buffer. For example, in Figure 4.5, the two outputs named \( y \) and \( z \) are associated with the buffer named \( \text{buf}_B \) of process \( \text{proc}_D \).
Chapter 4. Specifying Distributed Applications Using DAL and DAPL

![Diagram](image)

**Figure 4.5** Two outputs named $y$ and $z$ are associated with the same buffer $buf_B$.

### 4.1.1 Example Diagram of a Distributed Application

In general, the notation allows a many-to-many relationship between processes and buffers. That is, many outputs originating from only one process can be connected to many buffers, and many outputs originating from many processes can be connected to just one buffer. The diagram in Figure 4.6 of a small application depicts the several different ways to connect processes. This diagram shows examples of:

- a process containing zero or more buffers, e.g., proc_A has no buffers, and proc_C has two buffers;
- a process having zero or more outputs, e.g., proc_D has no outputs, and proc_A has three outputs;
- a process connected to several other processes, e.g., proc_A is connected to proc_B twice, and is also connected to proc_C;
- several processes connected to one process, e.g., proc_B and proc_C are connected to proc_D;
- a buffer associated with one or more outputs from the same process, e.g., buf_C1 is associated with one output from proc_A, and buf_B is associated with two outputs from proc_A; and
- a buffer associated with one or more outputs from different processes, e.g., buf_D is associated with outputs from proc_B and proc_C.

By using the graphic notation, as in Figure 4.6, one can visualise the structure and connectivity of the application quickly and separately from the application’s functionality. So the application developer can develop the application’s structure and functionality independently and does not need to worry about the functionality when developing the structure, nor worry about the structure when developing the functionality, nor in which order they are developed.
Figure 4.6  A small application showing how processes can be connected.

4.2 DAL: Distributed Application Language

The graphical specification is limited to the structure and connectivity and does not specify requirements such as the (names of) hosts on which the application and its processes are constrained to exist, the order in which processes must be created at the time of instantiating the application, nor the functionality and data structures of the processes. A complete specification of a distributed application is written using the text based languages DAL and DAPL.

The non-hierarchical version of the DAL language and grammar is presented in this chapter, and the extension to the hierarchical version is presented in Chapter 5. The complete DAL grammar is presented in Appendix A.

4.2.1 DAL Specifications

The DAL language is used to specify the connectivity of processes of a distributed application, a set of host machines on which these processes run, and the order in which they are created. A particular DAL specification has six parts, as shown in the first rule of the DAL grammar. These parts are:

- an application name,
- a machines clause,
- a buffers clause,
- an outputs clause,
- a creation clause, and
- a processes clause.
Chapter 4. Specifying Distributed Applications Using DAL and DAPI

The buffers clause and the outputs clause are extensions for handling hierarchy and are discussed in Chapter 5.

**Application Name.** The first part of the DAL specification is to provide a name for the application. This name must be unique with respect to all other applications loaded into the DAS system.

**Machines Clause.** The machines clause is used to specify a set of host names to define the scope of hosts associated with this application. That is, processes of this application are permitted to run only on these specified hosts. If this set is empty, then the list of hosts is as specified by the DAS system addhost command. Furthermore, the DAS system limits all applications to run on the hosts in this list. Hosts not in the list cannot be assigned any processes.

**Creation Clause.** The creation clause is an ordered list of process names, and provides a simple means to vary the default order of creating processes. Therefore, an appropriately ordered list ensures that a process is created and is ready to accept messages before a message is sent by the sending process.

The order in which processes of an application are created is dependent on both the order of process names within this list (L), and also the default order in which processes are declared within the processes clause (C) described below. The order is determined by using the following three rules where p and q denote processes, and L ⊆ C.

- p is created before q if p ∈ L, q ∈ L
- p is created before q if p ∈ L, q ∉ L, p occurs before q in C
- p is created before q if p ∈ L, q ∈ L, p occurs before q in L

**Example 1:** If six processes are declared within a DAL specification in the following order, say, p1, p2, p3, p4, p5 and p6, and the creation clause contains only one process name, say, p1, then the order in which these six processes are created when instantiating the application is p2, p3, p4, p5, p6, and p1. That is p1 is created last.

**Example 2:** As in Example 1, six processes are declared in the order p1 to p6, but instead, the creation clause contains a list of three process names, say, p3, p4 and p1. Therefore the order in which these six processes are created is p2, p5, p6, p3, p4 and p1.

**Processes Clause.** The fourth part of a DAL specification is a clause in which information about all processes of this application is specified. This clause is called a processes clause and consists of a set of sub-clauses, one for each process. Such a sub-clause is called a process clause, and consists of four parts:
Chapter 4. Specifying Distributed Applications Using DAL and DAPI

- a process name,
- a requires clause,
- a machines clause,
- a buffers clause, and
- an outputs clause.

The requires clause is part of the extension for handling process hierarchy and is discussed in Chapter 5.

**Process Clause: Process Name.** The first part of a process clause is used to specify a unique name of the process which must be unique in this application.

**Process Clause: Machines Clause.** The machines clause of a particular process is used to specify a set of host names. Each process may have a different set of host names. A set of host names defines the scope in which the process may exist, and it must be a subset of the host names listed in the application’s machines clause. For example, if the machines clause of process proc_1 contains, say, host_1 and host_2, and process proc_2 contains, say, host_2 and host_3, then proc_1 is restricted to exist on either host_1 or host_2, and proc_2 is restricted to exist on either host_2 or host_3. In addition, the application’s machines clause must, at least, contain host_1, host_2 and host_3 to conform to the DAL language, otherwise a parse error will occur during parsing and the DAL specification is not loaded into the DAS system.

The machines clause of a process is optional. For the case when this clause is missing or empty of host names, then the machines clause of the application is used by default. For example, if the machines clause of the application contains the following four host names: host_1, host_2, host_3 and host_4, and a process is specified without a machines clause, then this process can be allocated to host_1, host_2, host_3 or host_4.

**Process Clause: Buffers Clause.** The third part of a process clause is a buffers clause. It allows the programmer to specify the name and type of several buffers. For example, a process containing an integer buffer called buf_1 would be listed in the buffers clause as:

```
int buf_1
```

The DAL grammar allows multiple buffers of several types in a process. For example, a particular process might contain five buffers: buf_1, buf_2, buf_3, buf_4 and buf_5 of varying types such as:
int buf_1;
int buf_2;
int buf_3;
char buf_4;
char buf_5

**Process Clause: Outputs Clause.** The last part of a process clause is the outputs subclause to specify all outputs of a process to connect this process to other processes. The information required for each output is the type and name of an output, and the buffer to which this output is connected. The target buffer is identified by its name and the name of its process. Because buffer names are only unique in the scope of a process, the process name is also required to identify a particular buffer.

Formally, the DAL grammar requires the type and name of the output, the name of the destination buffer, and the name of the destination process. These three names are separated by a colon. For example, in Figure 4.6, process proc_A has three integer outputs called a, b and c. Outputs a and b are connected to buffer buf_B of process proc_B, and output c is connected to buf_C1 of proc_C. This is declared in the outputs clause of process proc_A as:

```plaintext
int a: buf_B: proc_B;
int b: buf_B: proc_B;
int c: buf_C1: proc_C
```

### 4.2.2 Validating Connectivity

The three output descriptions above conform to the DAL grammar. However, they must conform to the semantics of the DAL language too. That is, for a particular output description, the following must be true:

- The output type must match the type of the destination buffer. This is because the type of data sent by one process to another, must be inserted into a buffer of the same type.
- The destination buffer name must match the name of a buffer specified in the buffers clause of the destination process. This must hold in order to connect the output to the appropriate buffer at the time of application instantiation.
- The destination process name must match the name of another process.
- All buffers must be connected to at least one output.

The connectivity of an output and a buffer cannot be validated until information about both output and buffer is collected. However, an output and its associated buffer are specified within two different processes. Therefore, the connectivity information of
one process must be stored until the second process specification is parsed. As such, all connectivity information is stored during the parse phase, and later checked for correctness.

### 4.2.3 Examples of DAL Specification

Two examples of DAL specifications are presented in this section. Both examples are associated with the application diagrams depicted in Figure 4.4 and Figure 4.6. The first application is small containing two processes, two buffers, and two outputs. The second example contains four processes, four buffers, and six outputs.

**Example 1.** A DAL specification of the distributed application depicted in Figure 4.4 is listed in Figure 4.7. It completely describes structure and connectivity of the application, but the code includes specifications for host names and the order in which to create the processes.

```plaintext
application example_1
machines host_1, host_2, host_3
creation proc_1
processes
  process proc_1
  machines host_1, host_2
  buffers int buf_1
  outputs int a: buf_2: proc_2
end

process proc_2
machines host_3, host_2
buffers int buf_2
outputs int b: buf_1: proc_1
end

end
```

**Figure 4.7** An extended DAL specification of the diagram presented in Figure 4.4.

This DAL specification specifies an application called `example_1` in which all processes are restricted to three machines: `host_1`, `host_2` and `host_3`. The creation clause dictates that process `proc_1` must be created last.

The first process clause refers to the process named `proc_1`. It shows that `proc_1` can exist on either `host_1` or `host_2`, but excludes `host_3` by omission. This process has only one integer buffer, `buf_1`, and only one integer output, `a`. Its output clause indicates that this process is connected to another process called `proc_2` by means of its output `a` which is connected to buffer `buf_2`.
The second process clause refers to the process named proc_2. This process is very similar to proc_1. It may exist on either host_3 or host_2. It has one integer buffer, buf_2, and one integer output, b respectively. This process is connected to proc_1 because its output b is connected to buffer buf_1 of proc_1.

This DAL specification implies that, given the proper functionality of both processes, either process can send data to the other process. The data in this case is simply arrays of integers.

**Example 2.** A DAL specification that completely describes the distributed application of four processes depicted in Figure 4.6 is presented in Figure 4.8. This DAL specification lacks a machine clause for the whole application, a creation clause, and machine clauses for each process. So default values set by the DAS operator are used instead for the host names. For example, if five host names, say, h1, h2, h3, h4 and h5 have been assigned to the DAS system, then each process can be allocated to any one of these five machines.

```plaintext
application example 2
processes
  process proc_A
  outputs
    int a: buf_B: proc_B;
    int b: buf_B: proc_B;
    int c: buf_C1: proc_C
  end

  process proc_B
  buffers int buf_B
  outputs int a: buf_C2: proc_C, b: buf_D: proc_D
  end

  process proc_C
  buffers int buf_C1, buf_C2
  outputs int a: buf_D: proc_D
  end

  process proc_D
  buffers int buf_D
  end
end
```

**Figure 4.8** A DAL specification of the application diagram presented in Figure 4.6.
The default value for an empty creation clause is the list of process names in the order in which the processes are declared within the DAL specification. For example, proc_A, proc_B, proc_C and proc_D is the default list for the specification in Figure 4.8.

When the machines clause and creation clause of the DAL specification are omitted and the default values described are used, by including these clauses the designer can specify a set of hosts on which the application will run and also the order in which the processes are created. In addition, for each process, the designer may apply an individual restriction, i.e., the process may run on a host selected from a subset of these hosts. For example, say, process proc_A runs only on a host called h1, proc_B runs on h1 or h2, and the remaining two processes run on the default set of hosts. The order in which these four processes are created is, say, proc_D, proc_C, proc_B and lastly proc_A, i.e., in reverse order in which they are declared within the DAL specification. Given these requirements, the DAL specification in Figure 4.8 should be updated to the one in Figure 4.9.

```
application example_2
machines h1, h2, h3, h4
creation proc_D, proc_C, proc_B, proc_A
processes
process proc_A
machines h1
outputs
    int a: buf_B: proc_B;
    int b: buf_B: proc_B;
    int c: buf_C1: proc_C
end

process proc_B
machines h1, h2
buffers int buf_B
outputs int a: buf_C2: proc_C, b: buf_D: proc_D
end

process proc_C
buffers int buf_C1, buf_C2
outputs int a: buf_D: proc_D
end

process proc_D
buffers int buf_D
end
```

Figure 4.9 A DAL specification with machines and creation clauses.
4.3 DAPL: Distributed Application Process Language

In this section we consider the specification of process functionality. A formal way to specify a process is to use a programming language to write a program, that is, the program is the process specification. For example, the C programming language can be used to write a C program, an object module can then be compiled, and this object module can be used to create a process.

Using the C language, or some other contemporary programming language, to write one program for each process of an application appears to be adequate. However, such contemporary languages lack statements to describe interfaces to link several programs and the DAL specification together to form a complete specification of the distributed application. Therefore, a new language is required.

The DAPL language contains statements not only for specifying the process functionality, but also to describe an interface for linking to an application's structure described by the DAL specification. An interface is directly dependent on the process name, process buffers, and process outputs that are specified in the DAL specification. For example, if a DAL specification contains a process called proc_1 that has one buffer and one output, then the language used to specify the functionality of this process must also provide statements so that the programmer can specify such a process name, buffer, and output.

The DAPI language contains clauses to specify the program name, buffers, outputs and functionality. The body of the clause associated with such functionality consists of C declaration definitions and C function definitions, i.e., this body is C code. Therefore, the DAPI grammar actually is a super set of the C grammar.

4.3.1 DAPL Programs

The DAPL language is used to specify all processes of a distributed application as DAPL programs. Therefore, a complete specification of an application consists of a DAL specification, and a set of DAPL programs. Several DAPL programs can be placed within a single text file (the file extension is .dapl). The DAPL grammar for this language is presented in Appendix B.

For each process of a distributed application, the DAPL language is used to specify the following components of the process: the functionality, buffers, outputs, and structure definitions. A particular DAPL program has seven parts, as shown in Figure 4.10 and the first rule of the DAPL grammar (Appendix B).
program
   /* program name */

structures
   /* structure definitions */

buffers
   /* buffer clauses, one for each buffer */

outputs
   /* output clauses, one for each output */

pre_move_code
   /* zero or more C declarations, followed by */
   /* zero or more C statements */

post_move_code
   /* zero or more C declarations, followed by */
   /* zero or more C statements */

code
   /* C declarations and C functions, like a C file */

end

---

**Figure 4.10** The general format of a DAPL program.

**Program Name.** The first part of a DAPL program is a unique program name to distinguish it from all other programs in DAPL files that are to be used for a particular application. In addition, this program name must match the name of a process that is specified within the DAL specification so that the compilation system can associate the program with the appropriate process.

**Structures Clause.** The structures clause is optional and is used to specify structure definitions for use (by other statements) in the DAPL program. One or more structure definitions within another structure definition conforms to the DAPL grammar, which allows several levels of definition. The grammar permits such embedded definitions by means of rules that allow the type a data member to be either:

- a fundamental type such as char, int or float;
- an existing structure; or
- an in-line structure definition, i.e., a new definition.

Furthermore, another rule permits a data member to be either a single variable or an array. Figure 4.11 shows an example of the structures clause of a DAPL program.

Three structure definitions are shown in Figure 4.11: two definitions have the following structure tags address and employee. The employee definition contains four data members. The first is a simple unsigned integer. The pre-defined address structure definition is used for the type of the second and third data
members, i.e., the home address and postal address. The fourth data member is another structure to store salary information, its definition is called salary and is actually placed in-line within the employee structure definition.

The scope of these structure definitions is the whole DAPL program in which these definitions exist. Therefore, these structure definitions can be used to declare structures within any code segment of the DAPL program. However, for some applications, a structure definition should be in the scope of more than one DAPL program, not just one program. For example, if one process transmits data to another process, and the type of this data is a structure, then both DAPL programs must contain statements of the same structure definition.

The current DAPL grammar requires the programmer to place the same structure definition within both DAPL programs, i.e., the sending and receiving programs. As such, this grammar provides the essentials, and structured data can be sent from one process to another. However, a future version of the DAPL language can easily contain grammar to support include or inheritance mechanisms for the DAPL preprocessor to manage.

```
structures
struct address
{
    unsigned number;
    char street[20];
    char city[20];
    char country[20];
};

struct employee
{
    unsigned id;
    struct address home;
    struct address postal;
    struct salary
    {
        unsigned annual;
        unsigned medical;
        unsigned superannuation;
    };
};
```

Figure 4.11  Example structure definitions for a DAPL program.
Buffers Clause. The third part of a DAPL program is concerned with specifying the buffers. This clause is optional, and when omitted, it means that the process has no buffers. Such a process reading information from a database and then sending it via an output to another process requires no buffer statements. However, if a process has buffers, they must be specified in this clause.

The DAPL grammar specifies a type and name for each buffer and allows three optional code statements. These are executed in the event of data arriving at the process and being inserted into the buffer, or being removed from the buffer. Specifying a buffer requires the following five parts:

- a buffer type,
- a buffer name,
- a code clause for when data arrives,
- a code clause to be executed immediately before data is removed from the buffer, and
- a code clause to be executed immediately after data is removed from the buffer.

Buffer Clause: Buffer Type. The first part of a buffer clause contains the type of the buffer. This type can be either a simple type such as char, int or float, or a structure such as those in Figure 4.11.

Buffer Clause: Buffer Name. The name of the buffer immediately follows the buffer's type. It is a C identifier, and must be unique with respect to all buffers of the same process.

Example. Writing DAPL code for buffers that have no associated code clauses is as simple as declaring variables using the C language. The DAPL statements to declare the buffers clause for, say, two buffers b1 and b2, of type int and char respectively is shown in Figure 4.12. Each buffer specification is delimited by the keywords buffer and end.

buffers
buffer
    int b1
end
buffer
    char b2
end

Figure 4.12 Example of a buffers clause of a DAPL program.
Buffer Clause: Data Arrival Code Clause. The operational mechanism of a process is reactionary. That is, a process:

- blocks, waiting for data to arrive,
- wakes when data arrives,
- processes this data, and
- blocks, waiting for the next arrival of data.

Hence, the process must be able to block, detect incoming data, and invoke some code to process this data. These three capabilities are built into each process, and the DAPL programmer need only provide an optional stub of which is automatically invoked by the process when data arrives.

The code stub consists of optional C declarations such as local variables, followed by C statements for functionality. The DAPL programmer places this code in the data arrival code clause of the buffer clause.

Buffer Clause: Pre-Remove Code Clause. In addition to the stub of code associated with data arriving at a buffer, the DAPL programmer can provide another code stub which is invoked automatically and immediately before the data is removed from the buffer. This has been designed and implemented with buffers in order to provide a mechanism for the DAPL programmer to write code that is always executed before deleting data from the buffer. For example, without effecting the remove operation, the programmer can ensure that the buffer is not empty, or preprocess the buffer, or log information about this remove operation.

Buffer Clause: Post-Remove Code Clause. A third code stub, also written by the DAPL programmer, is similar to the pre-remove code. However, it is invoked immediately after deleting data from the buffer. That is, the programmer can use the post-remove code clause to write code to, say, log information about this remove operation.

Example. Figure 4.13 depicts the general format for specifying a buffer of a DAPL program called proc_1. Note that this general format does not require the programmer to write code to block the process, wake up the process on data arrival, or insert data into the buffer; such code is generated automatically by the DAS compilation system.

Outputs Clause. The DAPL language has been designed to support the programmer so that it is as simple as possible to send data from one process to another. Sending data means that consecutive data elements of the same type can be sent from one process to another as an array.
program
  proc_1
structures
  /* structure definitions */
buffers
  buffer
    buffer_type buffer_name
  data_arrival_code
    /* C declarations */
    /* C statements */
  pre_remove_code
    /* C declarations */
    /* C statements */
  post_remove_code
    /* C declarations */
    /* C statements */
end
  /* other kinds of clauses to be detailed below */
end

Figure 4.13 A generic form of a DAPL buffer statement.

The logical view of sending data is that an array of several elements is placed into one end of a communication channel. The DAS system takes this array and not only sends it to the other end of the communication channel, but also inserts this array into the buffer of the receiving process associated with the output end of the channel. In addition, the DAPL programmer is not required to write code for opening a socket, listening, or writing data onto the socket. The programmer need not be concerned because the DAS compilation system automatically generates such socket code.

In essence, the DAL and DAPL programmer must write at least three statements to send data from one process to another. One statement is in the DAL specification which describes how an output of the sending process is connected to a buffer of the receiving process. That is, this statement specifies the connectivity between the sending and receiving processes. The second statement is a declaration of the output of the sending process within the DAPL program. The third statement is in the DAPL program and calls the primitive function DA_send() provided by the DAS library.

The outputs clause of a DAPL program contains the specification of all outputs of the process being specified. It contains not only the name of the output, but also type and associated code. If a process does not have any outputs, the outputs clause can be omitted from its DAPL program. The outputs clause, as shown in the DAPL
grammar, commences with the keyword outputs which is followed by specifications for each output which consists of four parts:

- an output type,
- an output name,
- a code clause to be executed immediately before data is sent via this output, and
- a code clause to be executed immediately after data is sent via this output.

**Output Clause: Output Type.** The first part of an output clause contains the type of the output. This type, like the type of a buffer, can be either a fundamental type such as char, int or float, or a structure type.

**Output Clause: Output Name.** The name of the output immediately follows the output type. Like a buffer name, the name of an output is a C identifier and must be unique with respect to all outputs of the same process.

**Example.** DAPL statements to declare the outputs clause for two outputs \(x\) and \(y\) of type \(\text{int}\) and \(\text{char}\) respectively are listed in Figure 4.14. Each output specification is delimited by the keywords **output** and **end**.

```
outputs
  output
    int x
  end
  output
    char y
  end
```

**Figure 4.14** Example outputs clause of a DAPL program.

Two code clauses, for each output of a process, may be specified between the **output** and **end** delimiters. Both are optional, used to specify functionality, and are called the \textit{pre_send code clause} and the \textit{post_send code clause}. These two code clauses have been incorporated into the output clause of the DAPL language to provide a mechanism for the DAPL programmer to write code that is executed immediately before and after sending data via this output. The DAPL programmer may use these code stubs for some specific program related purpose such as logging information about message passing when using this output.

**Output Clause: Pre-Send Code Clause.** The statements of this code clause are executed immediately before data is placed onto the output, and are invoked automatically by the process. These statements are invoked immediately before the function \texttt{DA\_send()} is executed.
Output Clause: Post-Send Code Clause. This code clause also contains executable statements. These are invoked and executed immediately after data is sent. These statements are invoked immediately after the function DA_send() is executed.

Example. Figure 4.15 depicts the general format for specifying one output of some DAPL program called proc_1.

```
program
  proc_1
structures
  /* structure definitions */
buffers
  /* buffer clauses, one for each buffer */
outputs
  output
    output_type output_name
    pre_send_code
      /* C declarations */
      /* C statements */
    post_send_code
      /* C declarations */
      /* C statements */
end
  /* other kinds of clauses to be detailed below */
end
```

Figure 4.15 A generic form of a DAPL program consisting of one output.

This program also indicates that other clauses of a DAPL program can be placed after the outputs clause. Three additional DAPL program clauses are yet to be detailed. Two are associated with moving a process from one host machine to another. Moving processes facilitates flexibility of the distributed application as it allows a process to be moved from one host to another host and so that the application continues execution. The third clause is the code clause of a DAPL program which contains declarations and functions written in the C language by the programmer.

The concept of migrating a process from one machine to another is easily comprehensible, and has been implemented in some research operating systems. As the DAS system is currently implemented on a network of UNIX machines, and UNIX provides no support for migration, it is impossible. The DAS system supports process move instead of process migration. Process migration automatically saves and restores detailed state information such as all variable values and the function stack, whereas a move requires state information to be managed by the DAPL programmer.
In general, moving a DAS process from one machine to another involves terminating the process, and then creating another process on the destination machine. As UNIX does not support migration, the process created on the destination machine will commence in an initial state, which is not necessarily the same state as the terminated DAS process. For example, a DAS process of several outputs might terminate with some outputs open and others closed. The DAS system knows this state of the DAS process, but the UNIX system does not.

The DAS system, however, provides support for moving a process. That is, the outputs of the new process are opened or closed, by the DAS system, to match the open and closed outputs of the process it replaces. The disadvantage of a process move is that the DAPI programmer must write DAPI code to save data representing the process state to the file system or other storage just before the process terminates. To restore the state, the new process reads the state data. The DAPI programmer must write this code to restore the state of the moved process.

**Pre-Move Code Clause.** The statements of the `pre_move_code` code clause are executed immediately before the process move operation commences. These statements are written by the DAPI programmer to save process state data to files, so that it can be read by the new process to restore the process state.

**Post-Move Code Clause.** The statements in this code clause are executed immediately after the new process is created on the destination machine. These statements are written by the DAPI programmer to read process state data from files in order to restore the process state. This code is the counterpart of the code in the `pre_move_code` clause.

**Program Code Clause.** The final clause of a DAPI program is referred to as the program code clause in the DAPI grammar. The statements\(^1\) of this clause immediately follow the `code` keyword as shown in Figure 4.10. This clause allows the DAPI programmer to write the code specific to the functionality of the process. The DAPI parser has been designed and developed to parse code with respect to the DAPI grammar, and parse C code with respect to the complete C grammar. Therefore, the contents of the code clause is C declarations and C functions.

---

1. The DAPI programmer should be familiar with the C language because the statements within the code clause are C declaration and C function definitions, and this allows the programmer to specify the process functionality using the complete C language.
Each DAPL program, like a C program, contains one compulsory user function, \texttt{main()}, which is the first invoked function in a process. This \texttt{main()} function is DAPL's code clause, and may be used to create and initialise data structures and perform other initialisation tasks required when the process is created. A C program has the property that when its \texttt{main()} function finishes, then the process terminates. However, a DAPL process differs in that when the DAPI.\texttt{main()} finishes: it blocks, waits for data, processes the data, and, again, blocks for more data, and so on.

### 4.4 DAS Library User Functions

The DAS library contains system and user functions. DAS library system functions such as those to create buffers and outputs are unavailable to the DAPI programmer, whereas the user functions are available for sending data, accessing buffers, terminating an application process, or terminating the whole application. The DAS library contains six user functions:

- \texttt{DA\_send()}
- \texttt{DA\_remove()}
- \texttt{DA\_head()}
- \texttt{DA\_free()}
- \texttt{DA\_exit()}
- \texttt{DA\_exitapp()}

These six functions require few parameters with respect to sending data from one process to another, managing buffers, terminating processes and terminating an entire application. These functions are described below, and their usage is presented within the example in Section 4.5.

#### 4.4.1 \texttt{DA\_send()}

The user function \texttt{DA\_send()} is used to send a specified number of consecutive data elements of the same type to another process of the application. The data is sent to the second process via an output that connects the sending process to it. Therefore, the user function \texttt{DA\_send()} requires only three parameters: the name of the output, the address of the first data element, and the number of elements. The return value of this function is the number of bytes sent (on a UNIX implementation, this value is the one returned by the \texttt{write()} system call). This function’s prototype is:

\[
\text{int } \texttt{DA\_send(OUTPUT *name, void *location, int num);} \]

Example. Given an output named $x$ declared within a DAPL program, and an array of three integers such as:

```
int data[] = {123, 456, 789};
```

The DAPL statement to send these three integers to the process at the end of this output is:

```
DA_send(x, data, 3);
```

4.4.2 DA_remove()
The user function `DA_remove()` is used to remove an element from a specified buffer. DAS buffers are implemented as a FIFO queue, so this function removes the oldest buffer from the buffer. The element is made available to the calling function by returning a pointer to the element removed.

Logically, a buffer is a queue of variable sized arrays, all of the same type. That is, each buffer element is an array of data, of some length. For example, an integer buffer may contain three buffer elements, the first, an array of 10 integers, the second, an array of 20 integers, and the third, an array of 50 integers.

Each buffer element is a structure `buffer_data`, containing several data members such as:

- the array of data,
- the number of elements in this array, and
- the type of elements within the array.

The function `DA_remove()` requires only one parameter, the buffer name, and returns a pointer to a `buffer_data` structure. This function's prototype is:

```
struct buffer_data *DA_remove(BUFFER *name);
```

Example. Given that an integer buffer, named `buf`, contains several elements, and that the oldest element contains, say, 10 integers (in general, this number can be any non-negative integer), then `DA_remove()` may be invoked, most simply, by:

```
DA_remove(buf);
```

The above statement removes a buffer element, but the returned pointer is not used, and will cause some memory leakage. The following DAPL code removes the oldest element from the buffer, and uses the returned pointer to access the removed data. In this example, all integers in this element are printed to stdout.
struct buffer_data *element;
int *array, size, i;

element = DA_remove(buf);
array = element->array;
size = element->num_elements;
for(i = 0; i < size; i++)
    printf("%d\n", array[i]);

4.4.3 DA_head()
The user function DA_head() is like DA_remove() but the buffer element is not removed from the buffer. That is, this function allows the DAPI programmer to access the head element of the queue, without deleting the head element.

DA_head() requires one parameter, the buffer name, and returns a pointer to the buffer_data structure at the head of the queue. This function’s prototype is:

struct buffer_data *DA_head(BUFFER *name);

4.4.4 DA_free()
The purpose of the user function DA_free() is to deallocate memory to which was previously allocated to create a buffer_data structure. When data arrives at a process, the DAS system not only allocates a buffer_data structure to store the incoming data, but also inserts this structure into the appropriate buffer of the process.

In general, the DA_free() function should be used by the DAPI programmer when the buffer_data structure, obtained from invoking DA_remove(), is no longer required. This ensures that allocated memory is appropriately deallocated, so that no memory leaks occur. That is, for every invocation of DA_remove(), there should be an invocation of DA_free(). For example, the following statements are required, at least, to process data of which has both arrived at the process and stored in buffer called, say, buf.

struct buffer_data *element;

element = DA_remove(buf);
/* statements for processing the data element */
DA_free(element);
The DA_free() function requires only one parameter, the pointer to the buffer_data structure returned by the DA_remove() function. This function's prototype is:

```c
void DA_free(struct buffer_data *element);
```

### 4.4.5 DA_exit()

A process of an application can be forced to terminate when the user function called DA_exit() is invoked by that process. The DA_exit() function performs two tasks:

- change the state of the process to terminating, and
- sends a process shutdown message to the DAS system process, which informs the DAS system that the process intends to terminate immediately.

The DAS system process responds to this process shutdown message by performing two tasks:

- closing all outputs that are connected to or from the terminating process, and
- sending an exit message to the terminating process to instruct the process to continue the termination procedure.

A terminating process responds to an exit message, sent from the DAS system, by performing three tasks:

- sending a message to the DAS system that contains the exit status of the process, and also information indicating that process termination is impending,
- closing the port to the DAS system process, and
- in a UNIX implementation, invoking the system function called exit() using the exit status as parameter to terminate the process.

The function prototype for the DA_exit() function is:

```c
void DA_exit(int exit_status);
```

### 4.4.6 DA_exitapp()

A distributed application consists of several communicating processes. When an application completes its overall task (or tasks), then it can be terminated. Terminating an application means that all outputs throughout the application are closed, and all processes of the application are terminated.

An application can be forced to terminate when the user function called DA_exitapp() is invoked by any process of that application. The DA_exitapp() function only sends the exit application message to the DAS system process to inform the DAS system of the application's intention to terminate.
The function prototype for the `DA_exitapp()` function is:

```c
void DA_exitapp(int application_exit_status);
```

In general, the DAS system responds to an exit application message by changing the state of the application to terminating, and sending a process shutdown message to each of the application's processes. An application process responds to receiving the process shutdown message from the DAS system by simply invoking the `DA_exit()` library function. So, each process, in turn, invokes the `DA_exit()` function, in order to terminate the application. This two-phase shutdown is detailed in Section 7.3.4.4 Application Termination of the implementation chapter.

### 4.5 Example: Distributed Fibonacci

The distributed application presented in this section is an application of two connected processes which computes and displays the first 10 numbers of the Fibonacci sequence. These numbers are: 1, 1, 2, 3, 5, 8, 13, 21, 34 and 55. One process will display the nth Fibonacci number of the sequence, and send it and the previous Fibonacci numbers to the second process. The second process computes the Fibonacci number following the nth and sends it to the first process. The application will terminate after displaying the first 10 numbers.

Though this particular application is conceptually trivial, its specification, using the DAL and DAPL languages, clearly demonstrates the high level nature of the DAL and DAPL languages with respect to specifying application connectivity, sending data, and processing this data within the receiving process.

#### 4.5.1 Distributed Fibonacci: Diagram

The distributed application diagram of this Fibonacci application is depicted in Figure 4.16. The process called `display_Fibonacci` will display the nth Fibonacci number, and send the nth and (n-1)th Fibonacci numbers to the other process which computes the (n+1)th number. These last two numbers are sent simultaneously to the process called `compute_Fibonacci` via the output called `a`. The Fibonacci number computed by process `compute_Fibonacci` is sent to process `display_Fibonacci` via the output `b`. 
4.5.2 Distributed Fibonacci: DAL Specification

The DAL specification of this Fibonacci application is presented in Figure 4.17. The application name is Fibonacci, and the processes must run on machines called smaug, eldacar, or luthin. However, the machines clause of both processes further restrict which hosts can be used to allocate the processes. Process display_Fibonacci can be allocated to smaug or eldacar, and process compute_Fibonacci can be allocated to luthin or eldacar. The creation clause means that process display_Fibonacci will be created last.

```
application Fibonacci
machines smaug, eldacar, luthin
creation display_Fibonacci
processes
  process display_Fibonacci
    machines smaug, eldacar
    buffers int buf
    outputs int a: buf: compute_Fibonacci
  end

  process compute_Fibonacci
    machines luthin, eldacar
    buffers int buf
    outputs int b: buf: display_Fibonacci
  end
end
```

Figure 4.17 A DAL specification of the Fibonacci application.

The processes clause of this DAL specification contains specifications of two processes. Each process consists of one buffer, and one output. Both buffer names are buf, which is permitted because the scope of a buffer is the process in which the
buffer is declared. The output of process display_Fibonacci is called a, and its declaration means that integer arrays can be sent to buffer buf of process compute_Fibonacci. This output is used in the Fibonacci application to send the last two Fibonacci numbers as an array of size two. The second output, the one specified in process compute_Fibonacci is called b, and its declaration means that integer arrays can be sent to buffer buf of process display_Fibonacci. The computed Fibonacci number will be sent via this output b to process display_Fibonacci.

4.5.3 Distributed Fibonacci: DAPL Programs

The two processes in the DAL specification, display_Fibonacci and compute_Fibonacci, require two DAPL programs to specify their functionality. These are listed in Figure 4.18 and Figure 4.19. Process display_Fibonacci displays the next Fibonacci number, and sends the last two numbers to the process called compute_Fibonacci. Process compute_Fibonacci computes the next Fibonacci number (simply the sum of the last two numbers), and sends it to process display_Fibonacci.

The application is required to display the first ten Fibonacci numbers and terminate. Within the DAPL program for display_Fibonacci, a simple counter can be used to tally the numbers displayed, and the DAS library function DA_exitapp() can be invoked to terminate this application. This function is invoked when the first ten Fibonacci numbers have been displayed.

The control flow of this distributed application starts with the function main() of display_Fibonacci. This main() displays the first two Fibonacci numbers, and then sends them to the second process called compute_Fibonacci. The process compute_Fibonacci receives the last two Fibonacci numbers and sends their sum to process display_Fibonacci which displays this new Fibonacci number, and then sends the last pair of Fibonacci numbers to compute_Fibonacci. This is repeated until the first ten numbers of the Fibonacci sequence are displayed.

DAPL Program: display_Fibonacci. The DAPL program for the process display_Fibonacci is presented in Figure 4.18. This program is named display_Fibonacci and consists of three clauses: a buffers clause, an outputs clause, and a code clause.
```c
program
   display_Fibonacci
buffers
   buffer
      int buf
   data_arrival_code
      struct buffer_data *element;
      element = DA_remove(buf);
      prev_Fibonacci = next_Fibonacci;
      next_Fibonacci = get_int(element->array, 0);
      DA_free(element);
      printf("Fib(%d) = %d\n", ++tally, next_Fibonacci);
      if(tally < 10)
         DA_send(a, array, 2);
      else
         DA_exitapp(0);
   end
outputs
   output
      int a
   pre_send_code
      array[0] = prev_Fibonacci;
      array[1] = next_Fibonacci;
   end
code
   #define get_int(array, index) (*((int *)array)[index])
   int tally = 0;
   int array[2];
   int prev_Fibonacci = 1;
   int next_Fibonacci = 1;
   main()
   {
      printf("Fib(%d) = %d\n", ++tally, prev_Fibonacci);
      printf("Fib(%d) = %d\n", ++tally, next_Fibonacci);
      DA_send(a, array, 2);
   }
end
```

Figure 4.18 A DAPI program for the process display_Fibonacci.
program
    compute_Fibonacci
buffers
    buffer
        int buf
    data_arrival_code
        #define get_int(array, index) (*((int *)array)[index])
        struct buffer_data *element;
        int next_Fibonacci;

    element = DA_remove(buf);
    next_Fibonacci = get_int(element->array, 0) +
        get_int(element->array, 1);
    DA_free(element);
    DA_send(b, &next_Fibonacci, 1);
end
outputs
    output
        int b
end

Figure 4.19 A DAPL program for the process compute_Fibonacci.

The code clause of this DAPL program contains the specifications to start the application. It begins by declaring four program level\(^1\) variables (lines 29 to 32): tally to store the count of numbers displayed; an array of two integers which is used when sending the last two Fibonacci numbers to the other process; and the last two Fibonacci numbers, the second last in prev_Fibonacci, and the last in next_Fibonacci.

The main() function displays the first two Fibonacci numbers initially, 1 and 1, increments the tally as necessary, and sends two elements of the array to the other process via the output called a. This output is linked in the DAL specification to the buffer buf in the process compute_Fibonacci. However, this array contains no data before invoking DA_send(). A pre-condition to sending the last two Fibonacci numbers is that they are in the array. This is satisfied by statements in the proc_send_code clause of the output clause.

The buffers clause contains the specification for an integer buffer called buf. The next computed Fibonacci number is sent by the other process and stored into this

---

1. A program level variable is global to that program, but not to all programs.
buffer upon arrival. The DAS system stores this data into the buffer, and then automatically invokes the code within the buffer’s `data_arrival_code` clause (lines 7 to 18). Five tasks are performed when data arrives into this buffer. These are:

- line 9: remove the buffer element containing the next Fibonacci number from the buffer,
- lines 10 to 11: move the next Fibonacci number to the previous, i.e., extract the first integer (the next Fibonacci number) from the array of this buffer element, and store the second last Fibonacci number,
- line 12: deallocate this buffer element,
- line 14: display the next Fibonacci number, and
- lines 15 to 18: either send the last two Fibonacci numbers to the other process, or terminate the application after ten Fibonacci numbers are displayed.

**DAPL Program: compute Fibonacci.** The DAPL program for the process `compute_Fibonacci` is presented in Figure 4.19. This program consists of two clauses: a buffers clause and an outputs clause. It is unnecessary to include a code clause in this program because nothing needs to be done before this process blocks after instantiation, and it waits for two Fibonacci numbers to be sent by the other process. The process requires one output specified as an integer called `b` in the outputs clause of the DAPL program (line 19). The DAI specification links this output to the buffer `buf` in the process `display_Fibonacci`.

The buffers clause contains the specification for an integer buffer called `buf`. The last two Fibonacci numbers are sent by the other process and stored into this buffer upon arrival. This process sends the sum of these two integers back to the other process via the output `b`. The DAS system invokes the code in this buffer’s `data_arrival_code` clause (lines 7 to 15) when the two integers arrive. This code is used to send the sum and performs the following four tasks:

- line 11: remove the buffer element containing the last two Fibonacci numbers from the buffer,
- lines 12 to 13: extract the two Fibonacci numbers from the array and sum them to produce the next Fibonacci number,
- line 14: deallocate this buffer element, and
- line 15: send the next Fibonacci number to the other process.

**4.5.4 Distributed Fibonacci: Output**

A transcript containing DASH commands to create the Fibonacci distributed application described in Figure 4.18 and Figure 4.19, and also the output generated by creating and terminating this application is presented in Figure 4.20.
dash >>addhost smaug eldar car luthin elwing
dash >>addapp Fibonacci.dal
dash >>addprog Fibonacci.dapl Fibonacci
dash >>compile tibonacci
dash >>createapp Fibonacci
Fib(1) = 1
Fib(2) = 2
Fib(3) = 3
Fib(4) = 5
Fib(5) = 8
Fib(6) = 13
Fib(7) = 21
Fib(8) = 34
Fib(10) = 55
proc 'p1_0' of app 'Fibonacci', 1' terminated
proc 'p0_0' of app 'Fibonacci', 1' terminated
app 'Fibonacci', 1' TERMINATED

Figure 4.20 A transcript containing DASH commands to create the Fibonacci application, application output, and output generated when the application terminates.

This transcript contains the following three consecutive parts.

1. Five DASH commands to instantiate the application which:
   • add names of hosts to which the application instance is restricted;
   • add the DAL specification and DAPL programs to the DAS system;
   • compile the object modules for each process of the application; and
   • create an instance of the application.

2. The output of the application.

3. The output generated by the DAS system in response to a terminating application. This output shows that each process of the application has terminated, followed by the message that the whole application has terminated. The names p0_0 and p1_0 are the DAS names associated with the processes, the application name is Fibonacci, and the application’s unique ID follows this application’s name, e.g., 'Fibonacci, 1'.

4.6 Conclusion

Two new programming languages have been designed and developed called DAL and DAPL. In addition to these two new languages, a graphical notation has been developed to depict DAL specifications. Such notation provides a succinct means to
Chapter 4. Specifying Distributed Applications Using DAL and DAPL

visually appreciate an application’s connectivity, for example, the DAL specification in Figure 4.21 is depicted in Figure 4.22.

```
application minimum
processes
  process p1
    outputs int a: buf: p2
  end

  process p2
  buffers int buf
  end
end
```

Figure 4.21 A DAL specification that represents the minimum amount of code required to specify an application of two connected processes.

![Diagram of minimum application with processes p1 and p2, and buffer buf with int a and buf labels](image)

Figure 4.22 A distributed application diagram of the DAL specification of Figure 4.21.

The DAL language provides a means to quickly specify the connectivity of a distributed application using few statements that are easy to understand, relative to programming using UNIX sockets. For example, the DAL specification in Figure 4.21 is the minimum amount of DAL code required to specify two connected processes. It clearly indicates that the name of the application is minimum; the application consists of two processes: process p1 has an output called a that is connected to buffer buf of the process p2; and process p2 contains the buffer buf.

In general, the DAL language is used to specify:

- a unique name for a distributed application,
- the connectivity of all processes of an application,
- the order in which these processes are created, and
- the names of hosts to which the process allocation procedure is constrained to allocate processes.

On the other hand, the DAPL language is used to specify the functionality of each application process, and the data structures and interfaces required by these processes.
These two languages ensure that specifying an application’s structure or connectivity can be achieved independently from specifying the functionality of the application. This independence allows the design phase to be performed in a modular fashion. However, a DAL specification and its associated DAPL programs are not completely independent of each other, they require DAL and DAPL signatures to match (this is akin to matching a function call to the function’s prototype).

Sending data from one process to another is very easy using DAPL because one high-level statement is required for a process to logically place a number of array elements into one of its outputs using the DAS library function `DA_send()` such as:

```c
DA_send(output_name, array_name, num);
```

and this array is automatically transmitted to the receiving process at the other end of the output, and automatically placed into a buffer of this receiving process.

Six user functions from the DAS library support the developer with the means to send data, work with buffers and extract received data; and also two functions to terminate application processes and applications.

The DAL specification and DAPL programs that specify the Fibonacci application, presented in Section 4.5, show that network programming with respect to UNIX sockets has been completely replaced with high level abstractions of the DAL and DAPL languages. That is, network programming using the DAL and DAPL languages involves outputs, buffers and associated user functions such as `DA_send()` and `DA_remove()`. In addition to replacing the complexity of UNIX sockets programming, the amount of DAL and DAPL code is significantly less than that of a similar C program. For example, 77 lines¹ of code were written to specify the Fibonacci application using the DAL and DAPL languages. However, the amount of C code generated by the compilation system is 723 lines², plus 1500 lines (approximately) of the DAS header files and DAS library functions. Therefore, with respect to the code quality and quantity, the DAL and DAPL languages have a significant advantage over others such as the C language.

---

¹ 16 lines of DAL and 61 DAPL from Figure 4.17, Figure 4.18 and Figure 4.19.
² Those 723 lines of generated C code are listed in Appendix D.
5 Hierarchical Specifications of Distributed Applications Using DAL

The DAL and DAPL languages have been described in Chapter 4. These languages allow a programmer to formally specify distributed applications. However, the structure possible with these is non-hierarchical. The extension of hierarchical specifications is presented in this chapter.

In general, using hierarchical techniques to solve problems ‘greatly simplifies our understanding of the problem’ [Booch (1994)] and helps in managing the structural complexity of a system being developed. A hierarchical structure is important during development because it provides the means to build modular systems in which its modules can be independently designed, tested, debugged, modified and re-used.

The DAL and DAPL languages presented in Chapter 4 support re-use of DAPL programs, but not re-use of DAL specifications. For example, it is possible that one (or more) DAPL program can be part of the specification of several different applications.

Re-using DAL specifications is shown in this chapter to be possible when developing an application’s connectivity specification in a hierarchical way. In regards to the DAL and DAPI languages, such re-use is associated with specifying a process using an existing application. That is, an existing DAL specification of another distributed application is used as the specification for this process instead of a new DAPL program. For example, as shown in Figure 5.1, say application app_1 consisting of three processes exists and runs without error. A solution to a new problem requires another application of two processes, say app_2. One process of app_2 is specified using the DAPL language. However, if the functionality required for the second process of app_2 is equivalent to that of the existing application app_1, then the specification of app_1 could be re-used for this second process.

The DAL language and DAL grammar have been further developed to facilitate the use of existing application specifications to specify processes in new application specifications. In addition, an existing application specification can be used more than once. The essential concept associated with this re-use is that the functionality of a process, say PROC, can be equivalent to the functionality of an application, say APP. Therefore, the functionality of process PROC need not be specified as a DAPL program, but it can be specified as the existing DAL specification of application APP.
Chapter 5. Hierarchical Specifications of Distributed Applications Using DAL

The DAL specification of a distributed application is hierarchical when application re-use (as distinct from DAPIl program re-use) is involved. This hierarchical DAL specification takes the form of a tree, where each node of the tree is a DAL specification. For example, in Figure 5.1, a parent-child relationship exists between two nodes (app_2 and app_1) when a process (proc_2 of app_2) is specified by a DAL specification.

A leaf node of the hierarchy is a DAL specification in which each of its processes are specified as a DAPIl program. For example, app_1 of Figure 5.1 and all DAL specifications presented in Chapter 4 are leaf nodes.

A parent node may have one or more children, up to the number of processes in its DAL specification. There can be one child for each process specified using DAL.

The root node of a hierarchical DAL specification does not have a parent. The name of the entire application is the application name that is specified within the root node's DAL specification, e.g., app_2.

The DAL grammar presented in Chapter 4 is only sufficient for specifying a root node without children, i.e., a DAL specification for several connected processes. The DAL grammar presented within this chapter is extended to allow hierarchical DAL specifications. This is followed by a presentation of the complete DAL grammar and language. In addition, a graphical notation to depict a hierarchical DAL
specification is presented in this chapter. An example distributed application that is specified in a hierarchical fashion is presented, and this example is followed by the conclusions section.

5.1 Graphical Notation for Hierarchical DAL Specifications

The graphical notation for specifying a DAL specification presented in Section 4.1 includes several components such as symbols and labels for depicting a DAL specification in diagrammatic form such as the one in Figure 4.4.

The diagram of Figure 4.4 depicts the connectivity of processes with respect to only one DAL specification. However, a hierarchical DAL specification is a tree of one or more DAL specifications. Therefore, a distributed application diagram that depicts a hierarchical DAL specification must consist of more than one application diagram. There would be a diagram for each DAL specification, and each is a node of a tree. That is, a tree of application diagrams represents a hierarchical DAL specification.

The graphical notation for depicting a hierarchical DAL specification is slightly different from that for a non-hierarchical one. Additional graphical notation is required to represent two aspects:
- the parent-child relationship between a process of a parent DAL specification and the child DAL specification, and
- the interface to the child DAL specification.

5.1.1 Parent-Child Relationship

The graphics symbol used to represent such a parent-child relationship is a pair of dashed lines, as in Figure 5.1. These lines start at the process within the diagram of parent DAL specification and finish at the diagram of the child DAL specification. The pair of dashed lines are drawn to indicate that the specification of the process is that of the child DAL specification.

5.1.2 Interface

In addition to depicting the parent-child relationship, the interface of a child node must be part of the diagram. When an interface of the child node exists, it consists of at least one input buffer or one output. Each interface buffer belongs to one of the application's processes. They do not differ in functionality from the internal ones. The difference is that the data sent to an interface buffer is from another application. Likewise, each interface output is an output of a process, and behaves like all other outputs. However, such an output is used to send data to another application.
The graphical notation representing buffers and outputs in the interface of an application is similar to the notation representing internal buffers and outputs. That is, the symbols and labels for such buffers and outputs are the same: a buffer is represented as a square with a label, and an output is depicted as a labelled directed arc. However, there is a slight difference in notation associated with buffers and outputs to indicate that these buffers and outputs are part of the application's interface.

The graphical notation representing an interface output is a labelled directed arc, like other outputs. The directed arc starts at a process, but is not connected to any symbol. An output that is not connected to a destination process is an output of the application's interface. For example, Figure 5.2 shows two processes, p1 and p2, and three outputs, a, b and c. Output c is an interface output.

![Figure 5.2](image_url)  
Figure 5.2 A distributed application diagram containing an interface output c.

The graphical notation representing an interface buffer is a labelled square, similar to other buffers. A directed arc is connected to an interface buffer. However, to distinguish such a buffer from non-interface buffers, this directed arc does not start from a process within the application, and it is not labelled. That is, an interface buffer is associated with at least one output that is not connected to a source process. For example, Figure 5.3 depicts two processes, p1 and p2, and three buffers, x, y and z. Buffers x and y are interface buffers.

![Figure 5.3](image_url)  
Figure 5.3 A distributed application diagram containing interface buffers x and y.
Chapter 5. Hierarchical Specifications of Distributed Applications Using DAL  77

The interface of a process must match the interface of the application which is being used as the specification for the process. For example, in Figure 5.5, the interface of process Fibonacci_sequence matches the interface of application Fibonacci. That is:

- the number of outputs of the process must equal the number of interface outputs of the application,
- the type of each output of the process must be the same type as the corresponding interface output,
- the number of buffers of the process must equal the number of buffers of the application interface, and
- the type of each buffer of the process must be the same type as the corresponding buffer of the application interface.

In Figure 5.5, the interface of process Fibonacci_sequence consists of one buffer and one output, i.e., the integer buffer called length, and the integer output called seq. The interface of the application Fibonacci matches that of this process because it consists of one integer buffer and one output: the buffer called length and the output called seq in process create_sequence.

Although buffer length and output seq are associated with just one process, i.e., process create_sequence of Fibonacci, in general, buffers and outputs of an application interface may be associated with arbitrary processes of that application. Also, it is not necessary that names of buffers and outputs of an application interface match those of a process interface. In Figure 5.5, it is a coincidence that buffer length and output seq are associated with just one process, and that names match.

5.1.3 Example: Fibonacci Sequences

A new application, called Fibonacci_sequences, is to be created to display sequences of Fibonacci numbers. Each sequence starts at the first Fibonacci number and finishes at the Nth number. That is, for three values of N, say, 5, 7 and 10, the respective sequences to be computed and displayed by the new application are:

1 1 2 3 5
1 1 2 3 5 8 13
1 1 2 3 5 8 13 21 34 55

A simple design of this application consists of two processes: one process to send a value of N to the second process, this second process computes the Fibonacci sequence of length N and sends the sequence to the first process. On receiving a Fibonacci sequence, the first process displays the sequence, and then sends the next value of N to the second process to compute the next sequence. This continues until all values of N are exhausted.
A distributed application diagram representing the application Fibonacci_sequences is depicted in Figure 5.4. This diagram depicts two connected processes: display_sequence and Fibonacci_sequence. Process display_sequence performs two tasks. It sends a value of N to process Fibonacci_sequence via the integer output N. The second task is to display the sequence received from another process. The functionality of the process Fibonacci_sequence is to compute the first N numbers of the Fibonacci sequence and send this sequence, an array of N integers, to process display_sequence via output seq.

**Figure 5.4** A diagram of the Fibonacci_sequences application.

**Figure 5.5** A distributed application diagram where an application that displays one sequence of Fibonacci numbers is used in the new application.
The functionality of each process is specified as a DAPL program. However, to demonstrate hierarchy and re-use, the specification of process Fibonacci_sequence is not a DAPL program, but another application specification as in Figure 5.5.

From Chapter 4, an application that displays Fibonacci numbers is specified, and its specification can be re-used for the process Fibonacci_sequence. It is also altered so that it sends the sequence to another process instead of displaying the sequence. These minor adjustments involve altering its specification to allow the Fibonacci application to contain an interface of one buffer and one output. The size, N, of the Fibonacci sequence is sent to this buffer. The output is used to send a Fibonacci sequence to another process.

### 5.1.4 Converting Hierarchical to Non-Hierarchical DAL Specifications

A diagram of the hierarchical DAL specification for this new distributed application is presented in Figure 5.5. This diagram shows a tree of two nodes: the root node contains the diagram of the application Fibonacci_sequences, and the child node contains the diagram of application Fibonacci. The pair of dashed lines indicate the parent-child relationship between these two diagrams.

The hierarchical DAL specification depicted in Figure 5.5 can be transformed into a non-hierarchical DAL specification by drawing the re-used application specification in the place of the process being specified by this application. Figure 5.6 depicts the flattened version of the hierarchy depicted in Figure 5.5.

![Diagram](image)

**Figure 5.6** A non-hierarchical version of the hierarchical diagram in Figure 5.5.

Any hierarchical DAL specification can be transformed into a flattened DAL specification. The advantage of a flattened DAL specification over a hierarchical one is that the number and connectivity of all processes is clearly available. This information is essential to the DAS compilation system in order to generate appropriate C code, such as the correct number of C programs (one for each actual process). Furthermore, for this reason, the DAS compilation system converts a hierarchical DAL specification into a flattened DAL specification, as in Section 6.2.3.
5.1.5 Multiple Re-use of a DAL Specification

The parent-child relationship between a process of one DAL specification (parent) and another DAL specification (child) may be a many-to-many relationship. That is, one parent DAL specification can be related to many children, and one DAL specification can be used several times within a whole hierarchical DAL specification. For example, the root node of Figure 5.7 has three children, each one which has a different DAL specification. However, the DAL specification for application a5 is used multiple times: once for the specification of process p9, and again for the specification of process p11.

Figure 5.7 A diagram of a hierarchical DAL specification.
Although the DAL specification of application a5 is drawn only once within Figure 5.7, but is used multiple times, the implementation semantics is that multiple copies of the DAL specification are used, not just one. That is, process p9 and p11 are replaced in each case by a different instance of p12 and p13.

5.2 The DAL Language for Hierarchical DAL Specifications

The DAL language can be used to create non-hierarchical or hierarchical DAL specifications. Creating non-hierarchical DAL specifications has been detailed in Chapter 4, but here the focus is on specifying the parent-child relationship between applications, and the interface of a child application.

Three rules of the DAL grammar facilitate the production of a hierarchical DAL specification. These rules specify interface buffers, interface outputs, and which process requires an application’s DAL specification as its own specification.

The complete DAL grammar is listed in Appendix A. The starting rule of this grammar is DAL_SPECIFICATION, and two of its components are used to specify buffers and outputs of the application’s interface. This rule allows the user to specify:

- an application name,
- application machines,
- application buffers,
- application outputs,
- creation order of processes, and
- application processes.

Except for application buffers and application outputs, these components have been presented in Chapter 4. The application buffers and application outputs constitute the interface of an application. Specifying an interface is detailed in Section 5.2.1 Interface Buffers and Section 5.2.2 Interface Outputs.

Another rule directly associated with developing a hierarchical DAL specification is the rule called PROCESS. This rule is used to specify the following components of a process:

- a process name,
- an optional name of an application whose DAL specification is to be the specification of this process (required when specifying a child specification),
- process machines,
- process buffers (special buffer statements are not required when they are part of the application’s interface), and
- process outputs (special output statements are required when they are part of the application’s interface).
5.2.1 Interface Buffers

All interface buffers of a particular application are specified in the DAL specification of that application. They are declared within the buffers clause of the application immediately after the application’s machines clause.

The specification of an interface buffer consists of the following three parts:

- the type of the buffer,
- the name of the buffer, and
- the name of the process in which the buffer is declared.

The buffer name and process name are separated by a colon. For example, a DAL statement to specify an integer interface buffer called buf within process proc is simply:

```c
int buf:proc
```

In an application with several interface buffers, each is separated by a semicolon. For example, say an application’s interface contains three integer buffers and two character buffers, two of the integer buffers, b1 and b2, are in process p1, and the remaining buffers, b1, b2 and b3 are within a second process p2. Then, this application’s buffers clause would contain the following statements specifying these five interface buffers:

```c
int b1:p1;
int b2:p1;
int b1:p2;
char b2:p2;
char b3:p2
```

The DAL grammar allows interface buffers of the same type to be specified in a comma separate list. For example, the five statements above can be rewritten as the following two lists: a list of integer buffers, and a list of character buffers.

```c
int b1:p1, b2:p1, b1:p2;
char b2:p2, b3:p2
```

As presented above, the name of an interface buffer must be accompanied by the name of the process in which the buffer is defined. This pair of names is expected in order to conform to the DAL grammar which ensures that an interface buffer is uniquely identified. The scope of a buffer declaration is a process, and as such, the
same buffer name can be used in several processes. Using both the buffer’s name and the name of the process in which the buffer belongs, uniquely identifies the buffer. So although two integer buffers are named b1, above, the ambiguity is removed by appending the process name.

In addition to this naming constraint imposed by the DAL grammar, other constraints imposed by the DAL language are as follows:

- The name of a process must be valid; that is, a process with the same name must exist in the DAL specification. For example, if a DAL specification consists only of three processes, say, p1, p2 and p3, the following interface buffer statement is invalid because process proc does not exist.

```c
int buf:proc;
```

- The name of a buffer must be valid; that is, a buffer with the same name must be declared in the associated process. For example, if a process called p1 consists of two buffers, say, b1 and b2, the following interface buffer statements are valid.

```c
int b1:p1;
int b2:p1
```

However, the following statement is invalid because buffer buf does not exist within process p1.

```c
int buf:p1
```

- The type of buffer within the buffer clause of the application must match the type of the same buffer declared in the buffer clause of the process.
- A buffer cannot be declared more than once as an interface buffer.

### 5.2.2 Interface Outputs

Like interface buffers, all interface outputs of a particular application are specified in the DAL specification of that application. They are declared in the application’s outputs clause.

The specification of an interface output consists of three parts:

- the type of the output,
- the name of the output, followed by a colon, and
- the name of the process in which the output is declared.
For example, an application's outputs clause containing the following DAL statement specifies an integer interface output called \( x \) to be found in a process called \texttt{proc}:

\[
\text{int } x:\text{proc}
\]

The name of an interface output must be accompanied by the name of the process in which the output is declared, so that it is uniquely specified by its scope in the process. In addition, the DAL language constrains interface output statements as follows:

- The name of a process must be valid; that is, a process with the same name must exist in the DAL specification.
- The name of an output must be valid; that is, an output with the same name must be declared in the associated process.
- The type of the interface output must match the type of the same output that is declared in the output clause of the process.
- An output cannot be declared more than once as an interface output.

### 5.2.3 Example: Interface Buffers and Outputs

Interface buffers and outputs of the Fibonacci application are depicted in Figure 5.5, there is one interface buffer and one interface output in this application. The interface information can be mapped from this diagram to write the DAL statements.

The DAL specification for this Fibonacci application is presented in Figure 5.8 and contains statements regarding the application's interface. The buffer is described in the application buffers clause and the output is described in the application outputs clause. However, the destination process of the output called \texttt{seq} is unknown when developing this application. This is similar to not knowing the names or values of arguments within function calls when developing a function.

```plaintext
application
    Fibonacci
    ...
buffers
    int length:create_sequence
outputs
    int seq:create_sequence
    ...
end
```

**Figure 5.8** Interface statements for the application \texttt{Fibonacci} from Figure 5.5.
Chapter 5. Hierarchical Specifications of Distributed Applications Using DAL

When declaring an interface output, the names of the destination buffer and destination process are omitted from the output statement of the process clause. For example, the outputs clause of process `create_sequence` of application `Fibonacci` is:

```plaintext
outputs
  int a: buf: compute_Fibonacci;
  int seq::
```

Notice that the output called `seq` is not associated with the name of a buffer or a process. This means output `seq` is not connected to any process within the scope of this application, but to a process of some other application. Therefore, it specifies an interface output.

5.2.4 DAL Specification: Requires, Machines and Creation Clause

A parent-child relationship between a process defined by one DAL specification (parent) and another (child) is specified by using a requires clause within the parent specification. This means the process requires a named DAL specification instead of a DAPI program. The meanings of the `requires`, `machines` and `creation` clauses when a process requires a DAL specification are presented in this section.

**Requires Clause.** The requires clause of a process clause is used to change the way in which functionality of a process is to be specified. By default, a DAPI program is used. This clause specifies another DAL specification.

The requires clause of a process is optional, and if used, immediately follows the process clause. For example, from the hierarchical diagram in Figure 5.5, the process `Fibonacci_sequence` requires the DAL specification of application `Fibonacci`. Therefore, the DAL statements for this process contain a requires clause that refers to application `Fibonacci` as shown in Figure 5.9.

```plaintext
process
  Fibonacci_sequence
requires
  Fibonacci
buffers
  int length
outputs
  int seq: sequence: display_sequence
end
```

*Figure 5.9* DAL statements for the process `Fibonacci_sequence`. 
Machines Clause. It is possible that the machines clauses of process p and the DAL specification exist and differ\(^1\). The rule for resolution is that the machines clause of the required DAL specification overrides any other machines clause. This makes sense because the several processes in this DAL specification are specified to exist on particular machines, and these machines are named in this required DAL specification.

Creation Clause. The order in which application processes are created is affected by the creation clause of the application's DAL specification. However, many creation clauses might exist when a hierarchical DAL specification is used to specify the application, one creation clause for each DAL specification of the hierarchy.

A requires clause, with respect to the order in which processes are created, means that the order in which processes are created for the parent and child DAL specifications is not altered. However, the process, say p, that requires the child DAL specification is not created. In place of creating process p, all processes of the required application are created. For example, a parent DAL specification consists of five processes, say m, n, o, p and q, and the order in which these processes are created is m, n, o, p and q. Another DAL specification is required by process p. It consists of three process, say x, y and z, and the order in which these processes are to be created are x, y and z. Therefore, when the entire application is instantiated, the order in which these processes are created is m, n, o, x, y, z and q.

5.3 Example: Fibonacci Sequences

It is now possible to present DAL specifications and DAPL programs that specify a distributed hierarchical application. The distributed application in Figure 5.5 for computing and displaying several Fibonacci sequences is used to demonstrate the hierarchical DAL specification and DAPL programs.

The application is called Fibonacci_sequences. It consists of two processes at the root level of the hierarchy called display_sequence and Fibonacci_sequence. However, the diagram depicts that the functionality of process Fibonacci_sequence is to be the functionality of the application called Fibonacci. Therefore, process Fibonacci_sequence can be replaced by the processes of application Fibonacci. The number of processes in the entire application can be determined after such process replacement.

---

1. For example, the machines clause for process p contains, say, host_1, and the machines clause of the DAL specification might contain, say, host_2, host_3 and host_4.
Chapter 5. Hierarchical Specifications of Distributed Applications Using DAL

The entire application \texttt{Fibonacci\_sequences} is specified using two DAL specifications: one for the root node of the hierarchy, and one for the child node presented in Figure 5.10 and Figure 5.11 respectively. The DAL specification for the root node contains a statement on line 11 of Figure 5.10 indicating that process \texttt{Fibonacci\_sequence} requires a DAL specification.

```plaintext
1 application Fibonacci\_sequences
2 machines smaug, eldcar, luther
3 creation display\_sequence
4 processes
5    process display\_sequence
6        buffers int sequence
7        outputs int N:length:Fibonacci\_sequence
8    end
9
10    process Fibonacci\_sequence
11        requires Fibonacci
12        buffers int length
13        outputs int seq:sequence:display\_sequence
14    end
15 end
```

Figure 5.10 The DAL specification for application \texttt{Fibonacci\_sequences}.

```plaintext
1 application Fibonacci
2 machines smaug, eldcar, luther
3 buffers int length:create\_sequence
4 outputs int seq:create\_sequence
5 creation create\_sequence
6 processes
7    process create\_sequence
8        machines smaug, eldcar
9        buffers int length, buf
10        outputs
11        int a:buf:compute\_Fibonacci;
12        int seq:;
13    end
14
15    process compute Fibonacci
16        machines luther, eldcar
17        buffers int buf
18        outputs int b:buf:create\_sequence
19    end
20 end
```

Figure 5.11 The DAL specification for application \texttt{Fibonacci}.
The DAL specification of the child node in Figure 5.11 contains statements for specifying the application's interface. Therefore, this DAL specification must contain DAL statements based on application-buffers (line 3) and application-outputs (line 4) clauses. In addition, as a consequence of including an interface output, the statement in line 12 which declares this output indicates that it is not connected to a destination buffer or destination process.

The number of processes in the entire application is actually three. These are: process display_sequence from the root node, and processes create_sequence and compute_Fibonacci from the child node.

**DAPL Program: display_sequence.** The process display_sequence (Figure 5.12) not only dictates the length of a Fibonacci sequence to be created, but also displays the sequence when it arrives from another process. In addition, this process controls the number of sequences to be displayed.

The entire application is started by the code in lines 26 to 34 of Figure 5.12. The number of sequences is defined as thrcc (line 27). The length of each of these sequences is stored in an array (line 28). The integer n (line 29) is used to tally the number of requests for sequences. The main() function of this program initiates the first request. This is achieved by sending the length of the first sequence to another process via the output called N, after which this process blocks, waiting for a sequence to arrive from another process.

The data_arrival_code statements (lines 6 to 20) of the buffer called sequence are invoked when a sequence arrives at this process and is placed into this buffer. This code removes a sequence from the buffer and references it by means of element (line 11), displays each element of this sequence (lines 12 and 13), and either sends the length of the next sequence to be computed, or terminates the application because all sequences have been displayed (lines 17 to 20).

**DAPL Program: create_sequence.** The DAPL program for process create_sequence is presented in Figure 5.13. Upon receiving the length, N, of a Fibonacci sequence from the first process, it creates a sequence of the first N Fibonacci numbers, and then sends this sequence as an array of integers to the first process for displaying. This process creates the Fibonacci sequence by sending the last two Fibonacci numbers to a third process to which computes the next number in the sequence and returns it to this second process.
program
  display_sequence
buffers
  buffer
    int sequence
  data_arrival_code
    #define get_int(array, index) (*((int *)array)[index])
    struct buffer_data *element;
    int i;
    element = DA_remove(sequence);
    for(i = 0; i < element->num_elements; i++)
      printf("%d ", get_int(element->array, i));
    printf("\n");
    DA_free(element);
    if(n < SIZE)
      DA_send(N, sequence_length[n+1], 1);
    else
      DA_send(0);
outputs
  output
    int N
end
code
  #define SIZE 3
  int sequence_length[SIZE] = {5, 7, 10};
  int n = 0;
main()
{
  DA_send(N, sequence_length[n+1], 1);
}

Figure 5.12 A DAPL program for process display_sequence.

The process create_sequence reacts to two events: arrival of the length of
Fibonacci sequence, and arrival of the next Fibonacci number. The reaction to these
events is specified in the data_arrival_code clauses of buffer length and buffer buf.

The data_arrival_code clause of buffer length listed in Figure 5.13, obtains the length, N, of the sequence from this buffer (lines 8 and 9), allocates
enough memory to store the sequence of N integers (line 10), initialises the sequences
with the first two numbers (lines 13 and 14), and then sends the last two Fibonacci
numbers to another process, via output a (line 16).
program
  create_sequence
buffers
  buffer int length
data_arrival_code
  struct buffer_data *element;

  element = DA_remove(length);
  N = get_int(element->array, 0);
  Fibonacci = (int *)malloc(N * sizeof(int));
  DA_free(element);

  Fibonacci[0] = 1;
  Fibonacci[1] = 1;
  next = 2;
  DA_send(a, Fibonacci, 2);
end

buffer int buf
data_arrival_code
  struct buffer_data *element;

  element = DA_remove(buf);
  Fibonacci[next++] = get_int(element->array, 0);
  DA_free(element);

  if(next < N)
    DA_send(a, &Fibonacci[next-2], 2);
  else
    {
      DA_send(a, Fibonacci, N);
      free(Fibonacci);
    }
end
outputs
  output int a
end
  output int seq
end
code
  #define get_int(array, index) (*((int *)array)[index])

  int *Fibonacci;
  int N;
  int next;

main()
  {
  ...
end

Figure 5.13 A DAPL program for process create_sequence.
The `data_arrival_code` clause of the second buffer, `buf`, obtains the next Fibonacci number of the sequence from this buffer and stores it into the array of Fibonacci numbers (lines 23 and 24). If the sequence of `N` numbers is incomplete, the last two Fibonacci numbers are sent to process `compute_Fibonacci` via `output a` to compute the next number (line 28). When the sequence is finally created, the process sends the array of Fibonacci numbers to process `display_sequence` via `output seq` and also deallocates the array (lines 31 and 32).

**DAPL Program: compute_Fibonacci.** The DAPL program for process `compute_Fibonacci` is presented in Figure 5.14. Upon receiving the last two numbers of a Fibonacci sequence, this process computes the next number of the sequence and sends it back to the other process.

The functionality for this process is specified within the `data_arrival_code` clause of the buffer `buf`. The DAS library function `DA_remove()` is used to obtain the most recent received pair of Fibonacci numbers (line 11). The next Fibonacci number is computed by simply summing the last two Fibonacci numbers (lines 12 and 13), and then it is sent to process `create_sequence` via `output b` (line 15).

```dapl
1  program
2     compute_Fibonacci
3  buffers
4    buffer
5       int buf
6       data_arrival_code
7       #define get_int(array, index) {((int *)array)[index]}
8       struct buffer_data *element;
9       int next_Fibonacci;
10
11       element = DA_remove(buf);
12       next_Fibonacci = get_int(element->array, 0) +
13                      get_int(element->array, 1);
14       DA_free(element);
15       DA_send(b, &next_Fibonacci, 1);
16    end
17  outputs
18    output
19       int b
20  end
21
```

**Figure 5.14** A DAPL program for process `compute_Fibonacci`. 
5.4 Conclusion

The DAL language for specifying a distributed application has been presented in two parts. The description of the DAL language in Chapter 4 focused on specifying the connectivity of an application, the resultant specification being called a DAL specification. In Chapter 5, the specification technique was extended to a hierarchical specification of structures. That is, producing a DAL specification of a single application by means of a group of connected processes is presented in Chapter 4, and producing a DAL specification of an application by building on other applications to produce a hierarchical structure is presented in Chapter 5.

The DAL language has been designed such that DAL specifications can be produced in a hierarchical fashion. The resultant hierarchy is essentially a hierarchy of DAL specifications. In addition, the DAL language facilitates the re-use of existing hierarchical DAL specifications when creating a new DAL specification.

The DAL and DAPI languages support re-use of not only DAPI programs, but also DAL specifications. Re-using an existing DAL specification means that the functionality of a particular process is the combined functionality of the several connected processes which are specified within this re-used DAL specification. Furthermore, there is no reason for a DAPI programmer to write a DAPI program in order to specify the functionality of this process. Therefore, effort and time is greatly diminished by re-using DAL specifications.

The DAL language has been extended to allow a DAL programmer to produce hierarchical DAL specifications. Three clauses (sub-clauses of the process clause) have been added. The requires clause names an existing application as in the following extracted from Figure 5.10 (line 11).

\[ \text{requires Fibonacci} \]

This means that the functionality of process \text{Fibonacci\_sequence} is achieved by using the application called \text{Fibonacci}.

Other statements are used to specify interface buffers and interface outputs of this application as in the following extracted from Figure 5.11.

\[ \text{application Fibonacci} \]
\[ \ldots \]
\[ \text{buffers int length: create\_sequence} \]
\[ \text{outputs int seq: create\_sequence} \]
\[ \ldots \]
\[ \text{end} \]
This means that the interface of application Fibonacci consists of one integer buffer and one integer output. The buffer is called length and is declared in process create_sequence. The output is called seq and is an output of process create_sequence.

Specifying a distributed application using DAL and DAPL is described in Chapter 4 and Chapter 5. Such applications consists of many processes that are distributed over several workstations and also several distributed applications can be distributed over the network of workstations. A distributed application management system has been created to manage a set of distributed applications. This distributed application management system is called DAS and been designed and implemented as part of this research by the author. The DAS system is described in Chapter 6.
6 DAS: The Distributed Application System

The system, DAS, has been developed to manage several distributed applications. The complete set of features of DAS is presented in this chapter. Some of these features are associated with the life-cycle of an application, namely, compilation of the source code for a distributed application into object modules, instantiation of a distributed application, moving an application process from one workstation to another, and termination of the application.

The source code of an application is specified in the two languages, DAL and DAPL. These specifications must be loaded into the DAS system prior to invoking the compilation command.

The DAS system consists of three major components: a user interface; a compilation system; and a management system. The user interface to DAS is a text based shell, and is called DASH (Distributed Application SHEll). Twenty-five commands are available to the DAS operator for compilation and application creation and termination. The compilation system uses the DAL and DAPL source code specification of an application to generate object modules. The DAS system uses each object module to create one process per module when instantiating a distributed application. The same distributed application may be instantiated several times concurrently.

Management of a distributed application commences prior to instantiating the distributed application in that the DAS system requires information about the application such as the DAL specification and DAPL programs, and a set of host names on which the application's processes are permitted to run. The DAS operator may load host names, DAL specifications and a set of DAPL programs for each application using the DASH commands addhost, addapp and addprog. However, a feature of the DAS system allows one DAL specification to be associated with each of several different sets of DAPL programs. Therefore, it is possible to use one DAL specification to create several applications having the same structure, and each application can have a different set of DAPL programs to provide different functionality. For example, a particular DAL specification might specify the connectivity of, say, three processes: p1, p2 and p3. A DAPL programmer can not only write three DAPL programs for these three processes, but the programmer can also write three additional and different DAPL programs for the application.

Details of the DASH user interface, the compilation system and application management are presented in Section 6.1, Section 6.2 and Section 6.3 respectively.
6.1 DASH: User Interface to DAS

The functionality of the DAS system is accessed via DASH. Four groups of commands are provided:

- host name management;
- compilation;
- distributed application management; and
- internal DASHI commands.

These commands, their arguments and a brief description are presented in Table 6.1.

6.1.1 Host Name Management

The DAS system requires knowledge of the workstations on which the processes of an application are permitted to reside. The commands to manage the set of host names are: addhost, delhost and hosts. The addhost command is used to add several host names to the system. The host names are listed as arguments. For example,

```
addhost name1 name2 name3 name4
```

The delhost command is used to delete several host names from the system. The host names are listed as arguments. For example,

```
delhost name1 name2
```

The hosts command is simply used to display the current set of host names that are assigned to the DAS system. This command requires no arguments.

6.1.2 Compilation

Compiling object modules for a particular distributed application requires the specification of the application written in the DAL and DAPL languages. For each application, one file contains DAL source code, and one or more files contain DAPL source code. The commands associated with compilation are addapp, addprog, delapp, delprog, list, report and compile.

- The addapp command is used to load DAL specifications into the DAS system. The command arguments are the names of files in which DAL specifications exist. For example, if files app_1.dal and app_2.dal contain specifications for two applications, the commands to load both specifications are:
<table>
<thead>
<tr>
<th>DASH Command</th>
<th>Arguments</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>addhost</td>
<td>host1 host2 ...</td>
<td>Add several hosts to the systems.</td>
</tr>
<tr>
<td>delhost</td>
<td>host1 host2 ...</td>
<td>Delete several hosts from the system.</td>
</tr>
<tr>
<td>hosts</td>
<td></td>
<td>Display hosts.</td>
</tr>
<tr>
<td>addapp</td>
<td>file1 file2 ...</td>
<td>Add several DAL applications.</td>
</tr>
<tr>
<td>addprog</td>
<td>file1 file2 ... app</td>
<td>Add DAPI programs to the application.</td>
</tr>
<tr>
<td>delapp</td>
<td>name1 name2 ...</td>
<td>Delete several DAL applications.</td>
</tr>
<tr>
<td>delprog</td>
<td>name1 name2 ... app</td>
<td>Delete DAPI programs from the application.</td>
</tr>
<tr>
<td>list</td>
<td></td>
<td>Display names of applications and programs.</td>
</tr>
<tr>
<td>report</td>
<td>[ -a app</td>
<td>-p prog app ]</td>
</tr>
<tr>
<td>compile</td>
<td>name1 name2 ...</td>
<td>Compile several DAL applications.</td>
</tr>
<tr>
<td>createapp</td>
<td>name1</td>
<td>Create an application.</td>
</tr>
<tr>
<td>exitapp</td>
<td>appID ...</td>
<td>Terminate one or more applications.</td>
</tr>
<tr>
<td>status</td>
<td>[ appID ... ]</td>
<td>Display status table or status of several application.</td>
</tr>
<tr>
<td>creatproc</td>
<td>name appID [ host ]</td>
<td>Create a process and its connections.</td>
</tr>
<tr>
<td>creatproc_2</td>
<td>name appID [ host ]</td>
<td>Create a process, ignoring connections.</td>
</tr>
<tr>
<td>exitproc</td>
<td>p1 p2 ... appID</td>
<td>Terminate one or more processes.</td>
</tr>
<tr>
<td>move</td>
<td>p1 p2 ... appID [ host ]</td>
<td>Move a process.</td>
</tr>
<tr>
<td>open</td>
<td>o1 p1 ... appID</td>
<td>Open one or more process outputs.</td>
</tr>
<tr>
<td>close</td>
<td>o1 p1 ... appID</td>
<td>Close one or more process outputs.</td>
</tr>
<tr>
<td>help</td>
<td></td>
<td>This set of commands and descriptions.</td>
</tr>
<tr>
<td>exit</td>
<td></td>
<td>Terminate the system (also: quit, bye, x, q).</td>
</tr>
<tr>
<td>dash</td>
<td>file1 file2 ...</td>
<td>Evaluate DASH commands from file.</td>
</tr>
<tr>
<td>cd</td>
<td>[ dir ]</td>
<td>Change working directory.</td>
</tr>
<tr>
<td>pwd</td>
<td></td>
<td>Display the current working directory name.</td>
</tr>
<tr>
<td>ls</td>
<td>[ options ] [ file1 ... ]</td>
<td>List contents of directory.</td>
</tr>
<tr>
<td>;</td>
<td>UNIX command</td>
<td>Escape character to invoke a UNIX command.</td>
</tr>
</tbody>
</table>

Table 6.1  DASH commands.
addapp app_1.dal
addapp app_2.dal

or, one command may be used as follows:

addapp app_1.dal app_2.dal

- The `addprog` command is used to load DAPL programs into the DAS system. It also associates one or more programs with a DAL specification which was previously loaded using `addapp`. Each DAPL file may contain one or more process specifications. The leading arguments are the program file names, and the last argument is the name of a loaded application. For example, if the application called `example_1` consists of two processes, and the DAPL programs for the processes are in two files `p1.dapl` and `p2.dapl`, the DAPL programs can be loaded into the DAS system and associated with the application `example_1` using the following command.

```
addprog p1.dapl p2.dapl example_1
```

The name of the application, e.g., `example_1`, can be found explicitly specified within the DAL specification. The command above can be effectively replaced using the following two command.

```
addprog p1.dapl example_1
addprog p2.dapl example_1
```

- The `delapp` command is used to remove DAL specifications from the DAS system. The command arguments are one or more application names. For example:

```
delapp example_1 application_2
```

- The `delprog` command is used to delete one or more DAPL programs from the DAS system. The command arguments are one or more program names, and the last argument is the name of the application. For example, if the application called `example_1` consists of two processes and the associated DAPL programs are named `proc_1` and `proc_2`, the DAPL programs can be deleted using the following command:
delprog proc_1 proc_2 example_1

or, two commands may be used as follows:

delprog proc_1 example_1
delprog proc_2 example_1

NOTE: Replacing a DAPL program is simply achieved by deleting an existing one, and then loading another program. For example, the following sequence of commands load a DAL specification, two DAPL programs, and then replace one DAPL program with another specified in the file xyz.dapl.

addapp app_1.dal
addprog p1.dapl p2.dapl example_1
delprog proc_2 example_1
addprog xyz.dapl example_1

- The list command displays the names of all applications currently loaded into the DAS system. Following each application name is a list of names of all DAPL programs currently loaded and associated with this application. No arguments are required for this command. For example, the following two lines might be the output after invoking the list command.

example_1: proc_1 proc_2
application_2: p1 p2 p3 p4 p5

This output means that two DAL specifications called example_1 and application_2 have been loaded. Furthermore, several DAPL programs have been loaded, two loaded and associated with example_1, and 5 others loaded and associated with application_2.

- The report command confirms that details of DAL specifications and DAPL programs have been correctly extracted from the .dal and .dapl files and loaded into the DAS system. It displays details of a particular DAL specification using the -a option followed by an application name, or of a loaded DAPL program by using the -p option following by a program name and application name.
Example 1. The command:

    report -a example_1

displays the following kind of information.

    application name: example_1
    machine names: host_1 host_2 host_3
    application buffers: nil
    application outputs: nil
    creation processes: proc_1
    process names: proc_1 proc_2
    required application for process proc_1: nil
    required application for process proc_2: nil
    machine names for process proc_1: host_1 host_2
    machine names for process proc_2: host_3 host_2
    buffer data for process proc_1: int buf_1
    buffer data for process proc_2: int buf_2
    output data for process proc_1: int a buf_2 proc_2
    output data for process proc_2 int b buf_1 proc_1

Example 2. The command:

    report -p proc_1 example_1

displays the following kind of information.

    source filename:
        /home/rad/DA/examples/hierarchy/exl/pl.dapl
    program name:
        proc_1
    structures:
        -1 0
    buffers:
        b 'int' 58 159 -1 0 -1 0
    outputs:
        x 'int' -1 0 -1 0
pre move code offset and length:
   -1 0
post move code offset and length:
   -1 0
code offset and length:
   254 229

Pairs of integers, such as -1 and 0, and 58 and 159 in the above output are used
to specify the offset and length (in bytes) of a segment of code within the
specified source file. The pair -1 and 0 is used to specify no code: -1 for no
location, and 0 for the code length. In the above output, there is no code for the
structures clause, but there are 229 bytes of user code beginning at byte 255
(i.e., the offset + 1 = 254 + 1) in the file p1.dapl.

- The `compile` command is used to create object modules for one or more
distributed applications. After successful compilation, it is possible to
commence execution of a distributed application. The `compile` command
requires the names of applications that have already been successfully loaded
into the DAS system. For example, the following three commands load an
application called `app` and DAPL programs called `p1` and `p2` which are
associated with `app`, and then compile the application.

    addapp app.dal
    addprog p1.dapl p2.dapl app
    compile app

6.1.3 Distributed Application Management

The user manages distributed applications by means of the facilities provided by
DASH to create and terminate applications and display status information about such
applications. In addition, the operator can manage individual processes and
communication channels. For example, a blocked process that is waiting for input can
be terminated and then re-created on another host, whereupon it is restarted in the
state immediately prior to the termination. By this means a process can be moved
from one host to another. Similarly, the operator has the ability to open and close
channels.

The facilities supporting the management of a distributed application are in the
form of the commands `createapp`, `exitapp`, `status`, `createproc`,
`createproc2`, `exitproc`, `move`, `open` and `close.`
Chapter 6. DAS: The Distributed Application System

The `createapp` command is used to create an instance of a distributed application. As such, many instances of the same distributed application are possible which can run concurrently. The command takes a single argument, the name of a distributed application. Each executing application is distinguished by a unique system generated identification number called an application identifier.

When the command is issued, the DAS system creates the application in four steps:

- a host computer is selected for each UNIX process;
- each process is instantiated on its selected host using the appropriate object module;
- each communication channel is created; and
- the initialisation function `main` in each process is invoked.

The life cycle of a distributed application has three main stages: the above setup of the application by the DAS system, the processing activities performed by the processes of the application, and the termination of the application. An application can be terminated by either the DAS operator using the `exitapp` command, or calling the user library function `DA_exitapp()` by one of the application's processes. This library function and other library functions associated with the DAS system are presented in Section 4.4.

The `exitapp` command is used to terminate distributed applications and takes one or more application identifiers.

The DAS operator can view information about existing applications by using the `status` command. Information related to applications that have terminated is unavailable. The `status` command can be used with or without arguments, and the information it displays is presented and described in the following two examples.

**Example 1.** The `status` command without arguments displays information about the DAS system and a brief summary of each application. The first line of the following output shows the name of the DAS process, the machine on which this process is running, the port number by which application processes communicate with the DAS process, and the state of the DAS system such as RUNNING or TERMINATING. The second line shows a list of host names on which application processes may run. This is followed by a brief summary of each application containing unique application identifiers, names, and states.

```
das <host=host4, port=6001, RUNNING>
host1 host2 host3 host4
1D APP NAME STATE
1 example_A RUNNING
2 example_B RUNNING
```
Example 2. The status command is used with a list of application identifiers to obtain information about particular applications. For example,

```
status 1 2
```

displays the following kind of information about applications having 1 and 2 as their identifiers. For each application; its name, identification number, and state is displayed first and is followed by groups of several lines where each group contains information about each process. The first line of a group contains the system name of the process; the name of the machine on which the process is running; the process identification number from the host machine (e.g. a UNIX PID); the port number on which system level communication occurs; the exit status value, if any; the process state such as RUNNING, MOVING or TERMINATING. The second line of a group is a list of host names on which the process may exist. This is followed by information about outputs, namely their system names, associated port numbers, and states (open or closed) describing both ends of the communication channel.

```
example_A <id=1, RUNNING>
p0_0 <host=host1, pid=15304, port=6002, exit_status=0, RUNNING>
    host1 host3
    o0_0_0 port=6003 (open, open)
p1_0 <host=host2, pid=23870, port=6002, exit_status=0, RUNNING>
    host2 host4
    o0_1_0 port=35525 (open, open)

example_B <id=2, RUNNING>
p0_0 <host=host1, pid=15305, port=6003, exit_status=0, RUNNING>
    host1 host2 host3 host4
    o0_0_0 port=6006 (open, open)
p1_0 <host=host2, pid=12871, port=6004, exit_status=0, RUNNING>
    host2
    o0_1_0 port=6004 (open, open)
p2_0 <host=host3, pid=4880, port=6002, exit_status=0, RUNNING>
    host1 host2 host3
    o0_2_0 port=6005 (open, open)
```

### 6.1.3.1 Process Management

In addition to creating, terminating, and displaying the status of an application, the DASH shell provides the operator with commands to create, terminate, and move any process of an application. The commands are named `createproc`, `createproc2`, `exitproc` and `move`.
Chapter 6. DAS: The Distributed Application System

The commands createproc and createproc2 both create a process. The
difference between these two commands is concerned with creating the
communication channels that enter and exit the process. The createproc
command performs both actions, creating the process and all its communication
channels. The second command, createproc2, creates the new process without
the communication channels. In both cases, the host on which the process is created
can be specified as an argument. If the hostname is not specified as an argument, the
DAS system automatically selects a host from the list of hostnames associated with
the process in its DAL specification.

The exitproc command provides the DAS operator with the ability to
terminate a particular process of an application. Such an action might be required
when a process is performing in an unusual fashion and must be stopped. In order to
re-establish all processes of the application, another instance of the process can be
created by using the createproc command.

On occasions a particular host computer might need to be rebooted. For
example, system administrators may need to install kernel patches. Application
processes which are on a host that is being shutdown may be moved to another host
allowing interrupted operation. The move command provides the DAS operator with
the ability to move an application process from one host to another. In a system such
as UNIX which does not support process migration, moving a process from one host
to another is implemented by terminating the process and then creating another
instance on the second host. This is the case in the current implementation of DAS. As
a consequence of move, the open and closed communication channels after the move
replace the open and closed channels before the move, preserving the application’s
connectivity.

The connectivity of an application can be managed by the DAS operator by
invoking the commands open and close on communication channels identified by
the argument.

6.1.4 Internal DASH Commands

In addition to the commands for controlling applications, DASH also provides a small
set of internal commands. These are: help, exit, dash, cd, pwd, ls and an
escape character to access the operating system.

The help command displays a brief description of all DASH commands.

The exit command is to terminate the DASH interface and the DAS system. A
consequence of invoking the exit command is that all applications are terminated,
followed by the termination of the DASH shell and the DAS system.
In order to execute a sequence of DASH commands, they can be placed into a
text file and invoked by the `dash` command. Several file names can be arguments to
the `dash` command, and they are used in order of appearance.

Three internal commands are available to view and traverse the directories of
supporting operating system. These commands are `cd` (change working directory),
`pwd` (display the present working directory) and `ls` (display the contents of a
directory).

A DASH escape character is provided to allow the DAS operator to invoke any
command of the supporting operating system. This escape character is a semicolon
which is followed by the command string. For example, on a UNIX system

```
; ps -xg
```

causes the UNIX command `ps -xg` to be invoked.

### 6.2 Compilation System

A compilation system has been designed and implemented to convert the
specification of a distributed application into executable object modules as shown in
Figure 6.1. The specification of an application consists of a DAL specification\(^1\) and
DAPL programs written in the DAL and DAPL programming languages respectively.

It is possible that the specification of one application is modular, and so the
specification consists of several DAL specifications spread across several files. These
specification can be programmed independently by different programmers, and so it
is quite possible that two entities within the same scope have the same name. To
manage these ambiguities and scoping issues, the DAL specifications are
automatically rewritten by the *Canonical DAL Generator* into canonical DAL
specifications such that entities are renamed with unique identifiers.

The Canonical DAL Generator produces one canonical DAL specification for
each DAL specification of the hierarchy. These specifications can be combined by the
*Canonical DAL Integrator* to form one equivalent DAL specification where each
process described in this specification will map directly to an object module. This
single DAL specification and the associated DAPL programs are then translated in C
source files which used by the C compiler to generate the object modules.

---

1. Several DAL specifications are required when using a hierarchical DAL specification.
The compilation system consists of six sub-systems to generate the object modules. These sub-systems are:

- DAL Parser,
- Canonical DAL Generator,
- Canonical DAL Integrator,
- DAPI Parser,
- C Code Generator, and
- C Compiler.
These sub-systems and their relationships between one another are depicted in Figure 6.1. They are described in more detail in the following sections.

The compilation of object modules proceeds in the following manner. The DAL parser parses the files containing the DAL specifications. If parsing is successful, these DAL specifications are converted into canonical DAL specifications. The next stage of the compilation integrates the several canonical DAL specifications (including hierarchical DAL specifications) into a single canonical DAL specification and writes this specification into a file called final.dal (a file known to the DAS system only). This file contains the entire DAL specification of the application.

Prior to generating the C source code, the DAPL parser parses the DAPL programs. If this parsing is successful, the DAPL programs and the canonical DAL specification within the file final.dal are used by the C code generator to automatically produce several C files - one file for each process of the distributed application. In addition, a Makefile and Createfile are generated.

The final stage of this compilation procedure is to invoke a C compiler to produce object modules – one object module for each C file generated during the previous stage.

Prior to generating C programs, the DAL specification and DAPL programs are parsed to verify that:

- the DAL specification conforms to the DAL language, and
- each DAPL program conforms to the DAPL language,

and to extract essential data to:

- generate those C programs, and
- detect and report all errors such as references to undeclared variables, or inconsistent typing of outputs and buffers.

Here we are concerned with the outcomes of parsing, namely information extracted from DAL specifications and DAPL programs for syntactical verification and application construction.

### 6.2.1 The DAL Parser

The parser for the DAL language has been built using the flex and yacc tools [Levine, Mason and Brown (1992)]. The DAL grammar is presented in Appendix A. A DAL specification consists of the following lexical elements:

- **keywords**

  There are 18 keywords in the DAL language, half of which are names of fundamental types such as char or int. Currently, only fundamental types and structures are supported. The keywords are: application, machines,
creation, processes, process, requires, buffers, outputs, end, char, short, int, long, float, double, signed, unsigned and struct. Storage class specifiers like those within the C language, such as static, are not supported within the DAL language.

- **identifiers** A DAL identifier is the same as a C identifier. That is, an identifier commences with a letter or underscore, the remaining characters are either letters, digits, or underscores.

- **white spaces** All white space characters are ignored, these are: space, tab, vertical tab, form feed, new line.

- **comment lines** A DAL comment has the same style as a C++ comment, two forward slashes followed by any set of characters up to a new line.

The main purpose of the DAL parser is to validate a DAL specification with respect to the DAL language. However, a DAL specification must also conform to features of the DAL language that cannot be specified using a grammar such as rules ensuring that variables, used within expressions, are declared — and declared within the appropriate scope.

In order to generate C programs and validate a DAL specification, the parser collects essential information from the specification and also checks whether the specification conforms to requirements of the DAL language. These essentials and requirements are presented in the following two sections.

### 6.2.1.1 DAL Essentials

Information associated with an application is extracted from the DAL specification by the parser and stored in data structures. Information extracted is:

- the application name,
- the set of hostnames associated with the application,
- for each application buffer:
  - the buffer name,
  - the buffer type, and
  - the name of the process in which the buffer is declared,
- for each application output:
  - the output name,
  - the output type, and
  - the name of the process in which the output is declared,
- the list of process names associated with the creation clause of the application,
for each process:
  - the process name,
  - the set of hostnames associated with this process,
  - for each buffer of this process:
    - the buffer’s name,
    - the buffer’s type, and
  - for each output of this process:
    - the output’s name,
    - the output’s type,
    - the name of the destination buffer, and
    - the name of the process in which the destination buffer is declared.

This information is used to generate the C programs and to validate the DAL specification with respect to the DAL language. For example, the names of all processes of an application must be unique. Therefore, when a new process declaration is encountered, the parser checks whether the name was used previously.

6.2.1.2 DAL Requirements

Although a parser can be produced using the lex and yacc tools to determine whether a DAL specification conforms to the DAL grammar, not all DAL language features can be checked by such a simple parser. For example, undeclared variables are not detected by the grammar. Therefore, in addition to satisfying syntax, a DAL specification must satisfy certain rules of the DAI language, and so the functionality of the parser is extended to detect such language requirements. The DAL parser collects information from a DAL specification so that it can verify that:

- the machine names of an application are not duplicated,
- for each application buffer, its name, type and the process name in which it is declared is consistent. The parser verifies that the process name is one of the actual process names, the buffer name is one of the process’s actual buffer names, and the buffer type is the same type as the buffer in the process.
- for each application output, its name, type and the name of the process in which the output is declared is consistent. The parser verifies that the process name is one of the actual process names, the output name is one of the process’s actual output names, and the output type is the same type as the output in the process.
- the process names are not duplicated,
- the process names listed in the application’s creation clause are in the list of process names,
- the machine names of a process are in the list of application machine names,
- the output names of a process are not duplicated,
• the buffer names of a process are not duplicated,
• for each output of a process:
  • the destination process exists -- otherwise if the destination process does not exist, confirm that this output is an application output,
  • the destination buffer is part of the destination process, and
  • the type is the same type as the destination buffer, and
• for each buffer of a process:
  • there exists an output from another process that connects to this buffer -- otherwise if there are no outputs connected to this buffer, confirm that the buffer is an application buffer.

6.2.1.3 Parsing a Hierarchical DAS Specification
A hierarchical DAS specification allows for modular systems which are significantly more manageable than non-modular systems. A programmer writes a non-hierarchical DAS specification in a single text file containing information about all processes. On the other hand, a programmer writes a hierarchical DAS specification in more than one file, each corresponding to a node of the hierarchy.

Each node contains a DAS specification of one or more processes. For example, two nodes are shown in the hierarchy depicted in Figure 6.2. The parent node represents a DAS specification called a1, and the child node represents another DAS specification called a2.

![Diagram](image)

**Figure 6.2** A distributed application diagram depicting a parent-child relationship as a pair of dashed lines.
 Parsing a hierarchical DAL specification requires more work than simply parsing each file in the hierarchy because a naming conflict may occur when the process of a parent node is replaced by those of the child. For example, say the parent node consists of two processes, p1 and p2. Let process p1 be the process that is decomposed into several other processes specified within the child node. The conflict does not occur if the process names in the child node differ from p2. If the names are x, y and z, then the application really consists of processes x, y, z and p2 because p1 has been replaced by the three processes x, y and z. However, there is a conflict when one process within the child node has the same name as a process\(^1\) within the parent node.

Eliminating this conflict is possible by either requiring the user to rewrite the DAL specification using distinct names, or automatically converting the DAL specification(s) into a canonical form such that all names are guaranteed to be unique. The latter is the sound approach.

### 6.2.2 The Canonical DAL Generator

The conversion of a programmer’s DAL specification to a canonical form of that DAL specification is performed by the *canonical DAL generator*. The canonical DAL generator is one of the compiler’s components and is hidden from the user. It is invoked immediately after all DAL specifications are initially parsed.

For a particular application, the canonical DAL generator converts each file of DAL specifications to a file of canonical specifications. This ensures that all identifiers of applications, processes, buffers and outputs are unique by rewriting each identifier as a unique DAL system name (DAL name) which has a special format and is generated by the canonical DAL generator.

This format for uniquely naming applications, processes, buffers and outputs is essentially based on a prefix followed by a unique sequence of integers separated by underscores. The general format for such DAL names is:

\[
\begin{align*}
'a' & <\text{application ID}> \\
'p' & <\text{process ID}>'_'<\text{application ID}> \\
'b' & <\text{buffer ID}>'_'<\text{process ID}>'_'<\text{application ID}> \\
'c' & <\text{output ID}>'_'<\text{process ID}>'_'<\text{application ID}>
\end{align*}
\]

\(^1\) Excluding the process being decomposed because this process is substituted with all processes of the child node.
For example, if a hierarchical DAL specification consists of three application specifications, then their DAL names are:
\[ a_0, a_1 \text{ and } a_2. \]

If application \( a_0 \) contains, say, four processes, then their DAL names are:
\[ p_{0\_0}, p_{1\_0}, p_{2\_0} \text{ and } p_{3\_0}. \]

Regarding DAL names of buffers, if process \( p_{3\_0} \) contains, say, two buffers, then their DAL names are:
\[ b_{0\_3\_0} \text{ and } b_{1\_3\_0}. \]

Finally, if \( p_{3\_0} \) contains three outputs, the DAL names of these outputs are:
\[ o_{0\_3\_0}, o_{1\_3\_0} \text{ and } o_{2\_3\_0}. \]

### 6.2.3 The Canonical DAL Integrator

The canonical DAL integrator is a component of the compilation system which is automatically invoked after the generator has completed converting each DAL specification to a canonical DAL specification. The purpose of the integrator is to form a single canonical DAL specification by flattening a hierarchy of all canonical DAL specifications of an application.

Flattening integrates a child node into its parent node, continues until all children have been integrated, and results in a single node representing the entire application's DAL specification.

The output of the canonical DAL integrator is placed in a file called \textit{final.dal}. This file and the integration is part of the compilation procedure and hidden from the user (Figure 6.1).

At this stage, the compilation is not complete because three more components are waiting to be invoked, namely: the DAPL parser, the C code generator, and the C compiler. The C code generator requires the file \textit{final.dal} and the output of the DAPI parser to generator the C files, one for each object module. As such, the DAPL parser is the next component to be invoked.

### 6.2.4 The DAPL Parser

The parser for the DAPL language has been built using the flex and yacc tools. The DAPL grammar is presented in Appendix B.

A DAPL program consists of the following lexical elements:
Chapter 6. DAS: The Distributed Application System

- **keywords**
  There are 15 keywords, plus all the keywords of the C language, in the DAPI language. The keywords are:
  
  ```
  program       structures
  buffers       buffer
  outputs       output
  code          end
  data_arrival_code
  pre_remove_code   post_remove_code
  pre_send_code    post_send_code
  pre_move_code    post_move_code
  ```

- **identifiers**
  A DAPI identifier is the same as a DAL or C identifier.

- **white spaces**
  All white space characters are ignored, these are: space, tab, vertical tab, form feed, new line.

- **comment lines**
  The form of a DAPI comment is either the form of a DAL comment (C++ comment) or a C comment.

Like the DAL parser, the DAPI parser also collects essential information about the DAPI programs so that C code can be generated, and to confirm that the DAPI programs conform to the DAPI language.

### 6.2.4.1 DAPI Essentials

Information about a particular process is gathered from two separate files during compilation. This is because the specification of a process is written in two files: one containing connectivity information, and the other containing functionality information. The name of the process appears to be a way in which to identify the same process specification from both files. However, matching only names is insufficient. For example, names might match, but the number of buffers or outputs might differ.

DAL and DAPI program signatures must match to identify the same process specification within both files. A DAPI program signature consists of the program name, the number of buffers, the number of outputs, a list of its buffer types, and a list of its output types. A DAL program signature consists of the process name, the number of buffers, the number of outputs, a list of its buffer types, and a list of its output types. Two signatures match when respective details are the same. In addition, if process names within two signatures differ, then these signatures do not match and the remaining details can be ignored. If a match cannot be found, a compilation error has occurred and is displayed, and compilation ceases.
The information for the signature of a DAPL program is obtained from the DAPL program by the parser. The following information of a DAPL program is collected and stored:

- the file name that contains this DAPL program,
- the program name,
- the program structures (C structs),
- the offset and length of C code in the code clause,
- the offset and length of C code in the pre_move_code clause,
- the offset and length of C code in the post_move_code clause,
- for each buffer:
  - the buffer name and type,
  - the offset and length of C code in the data_arrival_code clause,
  - the offset and length of C code in the pre_remove_code clause,
  - the offset and length of C code in the post_remove_code clause, and
- for each output:
  - the output name and type,
  - the offset and length of C code in the pre_send_code clause, and
  - the offset and length of C code in the post_send_code clause.

The C code generator uses the file name, offset and length values to extract code from the specified file to generate C programs for which the C compiler later uses to produce object modules.

The program name is used as part of DAPL program signature. Structure definitions, from the structure clause of the DAPL program, are collected and stored. These are used when generating the C code for this DAPL program.

The name of each buffer and output is collected and stored because each name is used to declare a pointer variable within the C program. For example, a buffer name of abc, and an output name of xyz, are used to generate the following two pointer declarations during the C code generation stage.

```c
BUFFER *abc;
OUTPUT *xyz;
```

The type of each buffer and output is also collected and stored because this information constitutes the program signature.

The translation stage involves, in part, copying C code segments from the DAPL programs to the C programs being generated. These C code segments can be part of the following clauses of a DAPL program: data_arrival_code, pre_remove_code, post_remove_code, pre_send_code,
post_send_code, pre_move_code, post_move_code, and code. Therefore, it is essential to have information about these C code segments in order to locate them within the source file (the file containing the DAPL program). Hence, an offset, length and file name is stored. The offset and length specifies (in bytes) the location of the code segment from the start of the file, and the length of the segment. If any code segment is absent, offset = -1 and length = 0.

6.2.4.2 DAPL Requirements

Just as the DAL parser determines whether a DAL specification conforms to the DAL grammar and language, the DAPL parser determines whether a set of DAPL programs conform to the DAPL grammar and DAPL language. In addition to grammar checking, the DAPL parser verifies that:

- the program names (from the set of DAPL programs being parsed) form a set,
- the structure definitions conform to C structure syntax,
- the buffer names of a program form a set,
- the output names of a program form a set,
- each code segment from one of the following seven DAPL code clauses conforms to the body of a C compound statement, these DAPL code clauses are: data_arrival_code, pre_remove_code, post_remove_code, pre_send_code, post_send_code, pre_move_code, and post_move_code, and
- the code clause of a DAPL program conforms to a C program and may include a main() function. If the code clause is omitted or empty, the following default main() function is used.

```c
main()
{
}
```

Although, this default main() is without code, the process is still operational because this main() function is renamed to program_main() and invoked by the actual main() function that is generated during the translation stage.

6.2.5 The C Code Generator

The method employed to create executable object modules is to translate the canonical DAL specification and DAPL programs into another high level language for which a compiler already exists. The C language was selected, and the GNU compiler is used to create these object modules.
Chapter 6. DAS: The Distributed Application System

The C code generator produces not only C program source, but also a makefile and a text file called Createfile. A C program is generated for each process required to instantiate a distributed application. A makefile is also generated, it is used during the next stage of the compilation system to invoke the actual C compiler, gcc, and compile the object modules. Createfile is unique to the DAS system and contains information describing the order in which to create the processes at the time of application instantiation. This file is used by the DASH command createapp, but is otherwise hidden.

Generating these files is depicted using the dataflow diagram within Figure 6.3, and the contents of these files are presented in the following three sections.

![Dataflow Diagram](image)

**Figure 6.3** The C code generator and associated input and output files.

6.2.5.1 C Programs

Each C program produced by the C code generator contains 13 general components such as include directives, C code and functions extracted from the DAPI programs, and C code and functions automatically produced by the C code generator. These components and descriptions are presented in Table 6.2. Examples of C programs produced by the C code generator are listed in Appendix J.

6.2.5.2 Connectivity Data of the Process.

For each output (entering or exiting the process), the following data is stored within the generated C program.
<table>
<thead>
<tr>
<th></th>
<th>Component</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Include directives.</td>
<td>Directives to include C header files.</td>
</tr>
<tr>
<td>2</td>
<td>A unique process identifier.</td>
<td>A unique identifier of this process with respect to the DAS system consisting of the DAL name of the process and the DAS ID number of the instantiated application.</td>
</tr>
<tr>
<td>3</td>
<td>A unique well known port identifier.</td>
<td>A unique identifier of the well known port of the DAS system consisting of a port number and the name of the host on which the DAS system will run. The port provides the process with the capability of communicating with the DAS system, for example, to inform the DAS system that this process is terminating.</td>
</tr>
<tr>
<td>4</td>
<td>Socket descriptor and port number.</td>
<td>A socket descriptor and port number created by this process in order to receive messages from the DAS system. For example, if the DAS operator wishes to terminate the process, a message instructing the process to terminate is sent to the process via this port.</td>
</tr>
<tr>
<td>5</td>
<td>Process state.</td>
<td>The DAS state of the process such as RUNNING.</td>
</tr>
<tr>
<td>6</td>
<td>Connectivity data of the process.</td>
<td>See Section 6.2.5.2.</td>
</tr>
<tr>
<td>7</td>
<td>Output declarations.</td>
<td>A set of OUTPUT pointers, one for each output that leaves this process.</td>
</tr>
<tr>
<td>8</td>
<td>Buffer declarations.</td>
<td>A set of BUFFER pointers, one for each buffer within this process.</td>
</tr>
<tr>
<td>9</td>
<td>Structure declarations.</td>
<td>The set of user defined structures that are placed within the structure clause of this DAPL program.</td>
</tr>
<tr>
<td>10</td>
<td>Function prototypes.</td>
<td>Prototypes that are local to this DAPL program.</td>
</tr>
<tr>
<td>11</td>
<td>The C function main().</td>
<td>See Section 6.2.5.3.</td>
</tr>
<tr>
<td>12</td>
<td>The function program manager().</td>
<td>See Section 6.2.5.4.</td>
</tr>
<tr>
<td>13</td>
<td>C code from the DAPL code clause.</td>
<td>See Section 6.2.5.5.</td>
</tr>
</tbody>
</table>

Table 6.2 Components of generated C programs.
- The DAL name of the output.
- The DAL name of the source process of the output.
- A per process integer id number of the output. The underlying implementation enables many outputs between, say, process A and process B to be supported by opening and using one communication channel (based on only one pair of sockets). As such, an identification number is used to distinguish several outputs that use the same channel.
- A count of the number of open outputs that are using the same channel. If the count value is zero, then the channel may be closed.
- A boolean to indicate whether this output has been created.
- A pointer to a predefined C typedef called OUTPUT. This type is used to store information associated with a particular output, such as:
  - the actual socket descriptor integer,
  - the type and its size in bytes of the data accepted by this output,
  - the identification number of the destination buffer,
  - a pointer to the user function which is invoked immediately before data is sent via this output, i.e., the pre_send() function, and
  - a pointer to the user function which is invoked immediately after data is sent via this output, i.e., the post_send() function.

6.2.5.3 The C Function main().
main() consists of the following code:
- to initialise the process name and application ID;
- to initialise the host name on which the DAS process exists, and the port number for the DAS process;
- to initialise the process state;
- to create a socket and notify the DAS process of this socket so that it can communicate with this process;
- to create sockets to support communication channels for the inter-process communication of the processes that constitute the application;
- to create and initialise process outputs;
- to create and initialise process buffers;
- to send a message to the DAS process to inform DAS that this process has been initialised and is now running;
- to invoke the C function called program_main() which is actually the DAPI program function main() for this process. The C code generator rewrites the DAPI name main as the C function program_main because the main() function already exists. The function program_main() is invoked as soon as this process has been set up with appropriate sockets, output, buffers,
etc., as mentioned above. From the user’s point of view, the DAPL function main() is the first function invoked. The purpose of the DAPL main() function is to provide the DAPL programmer with the means of writing code to be executed at the commencement of the process;

• to invoke a DAS library function called program_manager() when the function program_main() has completed. program_manager() is described below in Section 6.2.5.4.

• to invoke the DAS library function called DA_exit() which informs the DAS system that process termination has commenced. The DAS system closes outputs connected to this process and requests the process to invoke the DAS system library function called _DA_exit(), and the UNIX exit() is soon invoked from within _DA_exit().

6.2.5.4 The Function program_manager().

program_manager() consists of a loop that continually iterates while the process is running. This loop consists of the following code fragments:

• Block on all sockets, waiting for data to arrive from another process either a process from within the same distributed application, or from the DAS system process.

• For each data arrival:
  • if the incoming data is on the DAS system socket, then process the message from the DAS system,
  • otherwise get the data from the communication channel; insert this data into the appropriate buffer; and invoke the function associated with this buffer in order to process the incoming data.

6.2.5.5 C Code from the DAPL Code Clause.

The remaining components of the generated C code are C code from the DAPL code clause such as variables and user functions written by the DAPL programmer, and some generated functions to create buffers and outputs. These components are:

• C code from the DAPL code clause such as main() and other programmed functions.

• Process move functions – pre_move() and post_move():
  • pre_move() is automatically invoked immediately before a process is moved from one host to another host, and
  • post_move() is automatically invoked immediately after the process has completed the move to the destination host.

• Buffer functions – data_arrival(), pre_remove() and post_remove(). Each of these three functions are prefixed with the name of
the associated buffer. That is, three buffer functions per buffer exist. If the buffer name is, say, \( x \), then the actual names of these three associated buffer functions are: \( x\text{-data\_arrival}() \), \( x\text{-pre\_remove}() \) and \( x\text{-post\_remove}() \):

- \( x\text{-data\_arrival}() \) is automatically invoked when data arrives and is inserted into the buffer called \( x \),
- \( x\text{-pre\_remove}() \) is automatically invoked immediately before data is removed from the buffer called \( x \), and
- \( x\text{-post\_remove}() \) is automatically invoked immediately after data is removed from the buffer called \( x \).

- Output functions - \( \text{pre\_send}() \) and \( \text{post\_send}() \). Each of these two functions are prefixed with the name of the associated output. That is, two output functions per output exist. If the output name is, say, \( y \), then the actual names of these two associated functions are: \( y\text{-pre\_send}() \) and \( y\text{-post\_send}() \):
  - the function \( y\text{-pre\_send}() \) is automatically invoked immediately before data is sent to another process via the output called \( y \), and
  - the function \( y\text{-post\_send}() \) is automatically invoked immediately after data is sent to another process via the output called \( y \).

- Automatically generated functions to create and instantiate buffers and outputs.

6.2.5.6 Makefile

The makefile is automatically produced by the C code generator. It is used during the compilation stage to invoke the C compiler (\texttt{gcc}) and therefore produce the object modules that are required to instantiate the distributed application. As an example, a makefile of a particular application is depicted in Figure 6.4.

```

# Makefile for the application 'example_app'

LIB /home/rad/DA/lib
INCLUDE=/home/rad/DA/include
INCLUDE_IPC=/home/rad/DA/include/ipc
LIBS = -lflow -lbuf -ldisp -lserver -ldiag -lnsl -lsocket
all: p0_0 p1_0 p2_0
p0_0: p0_0.o $(wildcard $(LIB)/lib*.a )
gcc -o p0_0 p0_0.o -I$(INCLUDE) -I$(INCLUDE_IPC) -L$(LIB) $(LIBS)
p1_0: p1_0.o $(wildcard $(LIB)/lib*.a )
gcc -o p1_0 p1_0.o -I$(INCLUDE) -I$(INCLUDE_IPC) -L$(LIB) $(LIBS)
p2_0: p2_0.o $(wildcard $(LIB)/lib*.a )
gcc -o p2_0 p2_0.o -I$(INCLUDE) -I$(INCLUDE_IPC) -L$(LIB) $(LIBS)
```

Figure 6.4 The makefile for the application called example_app.
The makefile is automatically generated per application because its content is determined by a particular application. It depends on the generated C programs of the application, and the DAL system names of the processes of that application. The file name of each generated C program is actually the DAL name of the process specified, plus the .c file extension. For example, if the process names are p0_0, p1_0, and p2_0, then the file names of the generated C programs are simply p0_0.c, p1_0.c and p2_0.c. After compilation, the following object modules would exist: p0_0.o, p1_0.o, and p2_0.o respectively.

Assuming no bugs created by the DAL and DAPI programmers, all object modules of an application exist at this point. Therefore, one might consider that executing each object module would result in the instantiation of the distributed application. However, the order in which each application’s process is created is important.

6.2.5.7 Createfile

A file called a Createfile is automatically produced by the C code generator. The DAS operator creates a distributed application by using the DASH command createapp. This command opens the createfile, and uses its contents to create the application.

The file is used to create the application’s processes in a specific order as specified (by the DAL programmer) in the creation clause of a DAL specification. However, if the DAL specification is a hierarchical one, then each node of the hierarchy may also have an associated creation clause. The createfile contains the combined creation clause of all those creation clauses throughout the hierarchical DAL specification.

The createfile is automatically generated per application because its content depends on the application. It also contains host names on which the processes are permitted to run, and the DAL names of outputs which must be created. For example, Figure 6.5 depicts a createfile for an application named example_app. Within a createfile, the order of processes specifies the order in which to create the processes.

```
example_app 3 host_1 host_2 host_3
p1_0 2 host_2 host_1 1 o0_1_0 p2_0
p2_0 2 host_3 host_2 2 o0_2_0 p0_0 o1_2_0 p1_0
p0_0 2 host_1 host_2 1 o0_0_0 p1_0
```

Figure 6.5 Createfile for the application called example_app.
The general format of a createfile has two parts. The first part is the first line of the file. This line contains the application name, the number of host names, and the host names. For example, the first line of the createfile in Figure 6.5 is:

```
example_app 3 host_1 host_2 host_3
```

which shows that the application is called `example_app` and that there are 3 hosts named `host_1`, `host_2` and `host_3` on which processes are allowed to run.

The second part of the createfile consists of several lines, one line for each process of the application, the order of which specifies the order in which these processes are to be created (as part of creating the application).

Specifying the order in which processes are created allows the designer to control the start-up sequence of an application. For example, say an application is designed to control an industrial task. Initialisation of this task may require interdependent sub-tasks to be created in a particular sequence to ensure successful start-up. The application processes are guaranteed to be created in the specified order because the DAS system sends a message to each process, in the specified order, to start each process. However, such a message is not delivered to the next process until a message from the current process is received by the DAS system indicating that the current process is running.

Each of line within the second part of the createfile contains:
- the DAL name of a process,
- the number of host names,
- the host names,
- the number of outputs leaving this process, and
- for each output, a pair of names:
  - the DAL name of the output, and
  - the DAL name of the destination process to which the output is connected.

For example, the second part of the createfile in Figure 6.5 is:

```
p1_0 2 host_2 host_1 1 o0_1_0 p2_0
p2_0 2 host_3 host_2 2 o0_2_0 p0_0 o1_2_0 p1_0
p0_0 2 host_1 host_2 1 o0_0_0 p1_0
```

which shows an application of three processes where `p1_0` is created first on either `host_2` or `host_1`, `p2_0` is created second on either `host_3` or `host_2`, and
finally p0_0 is created on either host_1 or host_2. In addition, process p1_0 has one output connected to process p2_0, process p2_0 has two outputs connected to p0_0 and p1_0, and process p0_0 has one output connected to p1_0.

6.3 Management of Distributed Applications

The management of applications by the DAS system can only occur if the DAS system process contains information about applications. Information must be internally available to the DAS system process before instantiation, after instantiation, and during the life time of the application instance. Details regarding information stored in the DAS process to manage applications is presented in the following sections.

6.3.1 Application Management: Pre-instantiation

When the DAS system process is created it contains no information about applications. The DAS process also has no information about which hosts may be used. Information about applications and hosts must be loaded into the DAS process by the DAS operator using DASH commands.

6.3.1.1 Host Name Management

The DAS system must have a list of host names. This list defines the set of hosts to which the DAS system may allocate processes. If this list is empty, it is impossible to instantiate an application. Furthermore, if an application instance is specified to exist on a set of hosts, this set must be a subset of the DAS system list of hosts.

6.3.1.2 Application Management

The DAS system must have information about an application before an application instance can be created. An application instance is dependent on the DAL specification and one or more DAPL programs. As such, data structures have been designed and implemented for the DAS system to support the storage of information extracted from DAL specifications and DAPL programs. This information structure is a one-to-many relationship, as is depicted in Figure 6.6.

The connectivity of several processes of an application is specified in a DAL specification and the functionality is specified in several the DAPL programs, one process specified by exactly one DAPL program. So, the DAL specification and an appropriate set of DAPL programs for an application must be selected and loaded into the DAS system prior to creating the application instance.

The steps to create an instance of an application are described in Section 6.1 using the addapp, addprog and createapp commands. Once the DAL
specifications and DAPL programs have been loaded, and the object modules exist,
then the operator can create (and start) an application instance by invoking the
creatapp command.

![Diagram](image)

Figure 6.6 Entity-relationship diagram of the DAS system, DAL specifications,
DAPL programs, applications, processes and outputs.

6.3.2 Application Management: Post-instantiation

From a user’s point of view, executing an application instance would seem to be the
only activity. A major goal is to provide this facility. The user may not even be aware
that the application consists of many processes. However, just as an operating system
must store information about processes in order to manage these processes, the DAS
system must be aware of an application instance as soon as this instance is created so
that it can be managed. The purpose of this section is to describe the DAS system with
respect to creating application instances, but only immediately after creating such
instances. Information about applications during their execution is described in
Section 6.3.3.

There is a one-to-many relationship between a DAL specification and
application instances. That is, it is possible to create many application instances that
run concurrently by using one DAL specification and related DAPL programs. This is
similar to creating many distinct UNIX processes using the same object module, for
example, creating many login shells. The relationships between all application
information are presented in Figure 6.6.
The `creatapp` command not only creates an application instance, it also stores details about this instance in the DAS system, such as the unique application ID, the state of the application, the locations of processes, i.e., the host names on which each process runs, and whether an output is open or closed.

Details about an application instance are stored in data structures of the DAS system. Three main data structures are used: one to store details about the application, another to store details about each process of an application, and a third to store details regarding each output of a process. There is a one-to-many relationship from an application instance to processes, and a one-to-many relationship from a process to outputs. These relationships are depicted in Figure 6.6.

Information about an application instance is stored within the DAS system for the life-time of that instance. During this period, some of this information is constant, but some may change. For example, the ID number of the application will not change, but the host name of a process will change when the process is moved from one host to another.

Although the `status` command displays information about one or more application instances, the DAS system holds more information than displayed by this command. Information about application instances, processes and outputs that is stored in the DAS system is detailed in the next three sub-sections.

### 6.3.2.1 Application Information

Information about an application instance is stored for the life-time of the instance, and is deleted when the instance terminates. Such information about an application is:

- the application name,
- the application ID,
- the application state,
- names of hosts to which processes of this application are restricted, and
- a reference to the information about each process of this application.

The hosts' names and process information are implemented as linked lists. Each element of the process's list is presented in the following section.

### 6.3.2.2 Process Information

Information about each process of an application consists of:

- the process name,
- names of hosts to which this process is restricted,
- a reference to the information about each output of this process (this information is described in the following section),
- the name of the host on which the actual process is running,
- a port number to which the DAS system can communicate with this process,
• the actual process identification number, e.g., the UNIX PID in a UNIX implementation,
• the process state,
• the exit status of the process,
• the name of a destination host, required when the process is moved from one host machine to another,
• the state of the process immediately before a move, and
• a list of process outputs that are open immediately before a process move operation. This list is required so that immediately after the move, all outputs associated with this process can be set to the same state (open or closed) as they were before the move. Each element of this list contains:
  • the name of an open output,
  • the two names of the output’s source and destination process.

The process state and information about open outputs is stored immediately before a move operation in the DAS system and is used to initialise the process to the same state that existed before the move\(^1\) operation. For example, if a process has been specified with three outputs, say, a, b and c, and a and b are open and c is closed before moving, then, immediately afterwards, the process will open only outputs a and b. That is, the state of communication of the application instance is invariant with respect to a move operation.

6.3.2.3 Output Information

Information about each output consists of:
• the name of the output,
• the name of destination process, i.e., the process to which the output enters,
• the port number,
• a boolean to indicate whether the source end of the output is open or closed, and
• a boolean to indicate whether the destination end of the output is open or closed.

6.3.3 Application Management: During Run-time

At run-time, the DAS system manages information about instances. That is, for the life time of an application the DAS system creates data structures, populates them, updates them, and finally deallocates them. The information stored within these data structures is the information described in Sections 6.3.2.1, 6.3.2.2 and 6.3.2.3.

---

1. Process migration is not part of UNIX, so the move operation is based on terminating the process and creating another that is in the same DAS state as was the terminated process. We speak here of these two UNIX processes as a single DAS process.
6.4 Summary

A software system called DAS (Distributed Application System) has been described in this chapter. It consists of three major components to manage many applications running concurrently. One component is the compilation system that compiles object modules from the DAL and DAPL specifications of a distributed application. Another component is the user interface to the DAS system. This interface is called DASH (Distributed Application system SHell) and is text based. It allows a DAS operator to invoke commands to load and unload DAL and DAPL specifications, create applications, manage applications that are running on a network, and view information about such applications. The third major component contains the data structures and operations pertaining to applications, which are intrinsic to the DAS system.

One of the main objectives at commencement of this research was to specify and create at least one distributed application connected in a specific fashion using DAL and DAPL and supported by compilers and libraries so that the programmer need not be concerned with coding for sockets or buffers or for managing events. It was intended to automate the creation of a distributed application by creating a system (now called the DAS system) to compile object modules, and then allocate processes on several computers. However, during development it became evident that the DAS system should be extended from managing one application to manage many. Hence a single DAS system is sufficient to manage a number of applications.

The current version of the DAS system manages many distributed applications as single entities, but the applications actually consist of several networked processes that are running on one or more machines.
7 Implementation of Distributed Applications

The concept of specifying a distributed application using a new language was formed when doing some (UNIX) network programming. Using the C language, the amount of code required to connect two processes with a pair of sockets was very large compared to the abstract notion that, say, process $P$ and process $Q$ are connected by a communication channel $Ch$. A pseudo code statement for such an abstraction can be as simple as: "$P.Ch.Q$" and as such, it was evident that gains of productivity and correctness could be made by using a new language for specifying connectivity of several processes.

The two new programming languages, DAL and DAPI, were designed and developed for the specification of distributed applications, DAL for connectivity and DAPL for functionality. A consequence of a new language is that a parser and a compiler/interpreter are required based on DAL and DAPL grammars. The formation of object modules is performed by translating the DAI and DAPL source into C code, and then using an existing C compiler to compile the required object modules, one module per application process.

Early in the project, an application was manually instantiated by using these object modules to create UNIX processes on one or more computers. This manual procedure was quickly replaced by an automated procedure.

The DAS system was developed from this automated procedure, to create and manage many such applications. A text based user interface called DASH was also built to provide a DAS operator with facilities to create and manage such applications.

Currently, the overall system is quite large, consisting of 97 files: 14 makefiles, 19 header files, 60 C source files, 2 flex files, and 2 yacc files. These files consist of around 17000 lines of source code, 51 structure definitions and 284 function prototype declarations in the header files. The makefiles create 57 object modules and place them into the following 11 libraries: libbuf.a, libdal.a libdapl.a, libdas.a, libdash.a, libdisp.a, libflow.a, libdp.a, libserver_sd.a, libstrbuf.a, and libtypetable.a.

Implementation Overview. The distributed application system (DAS) presented here is an outcome of a successful UNIX implementation of the following components: the DAL and DAPL languages and their parsers, the compilation system, the DASH user interface, and the data structures and messages that are internal to the DAS system.
Parsers for the DAL and DAPL languages are built using the flex and yacc compiler tools. DAPL programs contain functionality which is written using the C language, and, as such, the DAPI parser can parse DAPL statements and C statements too. The tasks performed by the DAL and DAPI parsers are presented in this chapter.

A distributed application is based on message passing, so a DAPI program must consist of statements to send data to another process running on the network via a communication channel, and access buffers containing data that has been sent from another process. Implementation details of buffers and outputs such as definitions of the BUFFER and OUTPUT types, and the automatic creation of buffers and outputs are presented in Section 7.2. Logically, a message sent from one process to another consists of an array of data, a logical data packet (LDP). However, sending a message requires additional information, such as the number of elements within the array, which is combined with the LDP to form an internal data packet (IDP). Details of the LDP and the IDP are presented in Section 7.2.

Implementation details of the DAS system components such as the user interface DASH and the compilation system are presented in Section 7.3.1 and Section 7.3.2. Hidden from the programmer and user are internal DAS data structures, DAS functions, and DAS messages. There are numerous DAS data structures for: storing information extracted from the DAL specifications and DAPL programs by the parsers; or storing information about running applications. DAS data structures are presented in Section 7.3.3. DAS messages used by the DAS system and application processes during particular events such as the creation or termination of an application, their details are presented in Section 7.3.4.

7.1 DAL Language Implementation

The DAL language described in Chapter 4 and Chapter 5 is used to form a hierarchical DAL specification of a distributed application. Its grammar is presented in Appendix A. The dataflow diagram in Figure 6.1 shows components of the DAS compilation system such as the DAL parser and DAPL parser.

The DAL parser has been implemented using flex and yacc. It must perform two tasks. The first is to extract information from each DAL specification in the hierarchy and store this information into several data structures1. The second task performed by this parser is to indicate whether the DAL specification is valid with respect to the DAL language. The parser does this by returning a pointer whereby:

---
1. The DAL data structures are presented in Appendix C.
• a NULL value indicates an invalid DAI specification, and
• a non NULL value not only indicates a valid specification, but also refers to the data structures containing the extracted information.

Referring to Figure 6.1, this extracted information is later used by the canonical DAL generator to produce files containing canonical DAL specifications which are then flattened by the canonical DAL integrator to produce a non-hierarchical DAL specification to finalise the actual processes and their connectivity.

Flattening a Hierarchy. A non-hierarchical DAL specification is produced by combining a parent DAL specification and child DAL specifications to form one DAL specification. This flattening procedure continues until all parent and child specifications are combined. The resultant specification is non-hierarchical because all children have been combined recursively into the root node.

This procedure completely replaces a process specification of the parent DAL specification, say P, with the specifications of all processes of the child DAI specification, say C. The procedure also ensures that the lists of machine names, and the list of process names within the overall creation sequence are updated correctly.

After flattening is complete, the resultant non-hierarchical DAL specification is written to a text file called final.dal in which DAL identifiers such as buffer names have been rewritten as DAS system names by the compilation system. For example, the hierarchical DAL specification of the Fibonacci_sequences application in Chapter 5 (Figure 5.10 and Figure 5.11) is transformed by the compilation system into a non-hierarchical DAL specification. The flattening procedure in this example replaces the specification of process Fibonacci_sequence of application Fibonacci_sequences with the specifications of the two process, create_sequence and compute_Fibonacci, from application Fibonacci. The resultant non-hierarchical DAL specification is presented in Figure 7.1 which also shows:

• DAS system names such as a0 and p0_0 (in the format discussed in Section 6.2.2);
• that interface buffers and outputs have been removed;
• a complete list of process names in the creation clause;
• an empty requires clause in each process (implying a non-hierarchical DAL specification); and
• a complete list of machine names in the machines clause of each process.
7.2 DAPL Language Implementation

The specification of each process in a non-hierarchical DAL specification must be a DAPL program. However, a hierarchical DAL specification is formed when the specification of a process conforms to the application to be used to replace that process.

```
application a0
machines smaug, eldacar, luthin
buffers
outputs
creation pl_1, p0_1, p0_0
processes
  process p0_0
    requires
      machines smaug, eldacar, luthin
      buffers int b0_0_0
      outputs int o0_0_0:b0_1_0:p0_1
    end
  end

  process p0_1
    requires
      machines smaug, eldacar
      buffers int b0_1_0;
      int b1_0_1
      outputs int o0_0_1:b0_1_1:p1_1;
      int o0_1_0:b0_0_0:p0_0
    end
  end

  process pl_1
    requires
      machines luthin, eldacar
      buffers int b0_1_1
      outputs int o0_1_1:b1_0_1:p0_1
    end
end
```

*Figure 7.1* The non-hierarchical DAL specification produced by flattening the hierarchical DAL specification in Figure 5.10 and Figure 5.11.

The DAPL language is used to specify five major features of a process: a name, data structures, functionality, buffers and outputs. The data structures and functionality are specified using statements from the C language. The process name, buffers and outputs are specified using DAPL statements, which do not belong to the C language.
That is, the DAPL grammar is C grammar plus additional grammar rules, and is presented in Appendix B.

The DAS compilation system uses all statements in a DAPL program to generate equivalent C source, which can be compiled by an existing C compiler to produce an object module.

The DAPL parser has been implemented using the flex and yacc compiler tools. Like the DAI parser, it performs two tasks. The first is to extract information from each DAPL program in the source file and store it in several data structures. The second task performed by the parser is to indicate whether the programs are valid with respect to the DAPL language. The parser does this by returning a pointer whereby:

- a NULL value indicates an invalid DAPL program, and
- a non NULL value not only indicates valid programs, but also refers to a linked list where each list element contains data structures of the information extracted from a DAPL program.

Referring to Figure 6.1, this extracted information and the generated non-hierarchical DAL specification (within the file final.dal) is later used by the C code generator to produce the C source, Makefile, and Createfile whereby the C compiler generates the executable object modules.

The structures used by the DAPL parser to store extracted information are presented in Appendix D. Implementation details of process buffers, process outputs and DAS data packets are described in the following sections.

### 7.2.1 Implementation of Process Buffers

A process of a distributed application can have zero or more buffers. For example, a process that reads data from files, processes this data and sends results to another process via an output has no need of a buffer.

Logically, the DAPL programmer views a buffer as a queue. Data sent from one process to a second process via an output is placed into a buffer. This buffer is not only connected to the output, but is also part of the second process. Each buffer element is an array of some type: either a fundamental type or structure. That is, data sent from one process to another is actually an array of like typed elements. In addition, the number of elements per array is dependent on the number of data elements sent by the sending process.

The DAPL programmer is provided with three buffer functions from the DAS library: DA_remove() to remove the head element of the queue, DA_head() to access the head element of the queue, and DA_free() to deallocate the memory allocated to a buffer element. Buffers are automatically created and initialised to be empty at the commencement of execution of a process, and therefore, there is no need
for a DAPL programmer to code the creation of a buffer, and such library functions for buffer creation are not provided.

In addition to these three library functions, the DAPL programmer can provide three other buffer routines to suit a particular application: one routine is invoked when data arrives at the buffer, another is automatically invoked immediately prior to executing DA_remove(), and the third is automatically invoked immediately after executing DA_remove(). The first routine is the one which is invoked automatically to act on the incoming data. Although the other two routines can be incorporated in the DAPL source code before or after every DA_remove() invocation to perform user defined tasks before or after every remove of a particular buffer, it is reasonable to place such code that is associated with a buffer at a single location within the source. The location of the code for these buffer dependent routines is within the buffer statement.

The specification of an integer buffer, say b, using the DAPL language can be as simple as the following statement.

```
buffer
  int b
end
```

Reference to this buffer within the DAPL program is by the buffer's name, i.e., b. For example, a DAPL statement to remove an element from buffer b can be:

```
DA_remove(b);
```

Three C data structures supporting these DAPL buffers are defined within the header file called buffer.h. These structures are hidden from the DAPL programmer, and are described below.

**Type BUFFER.** The type called BUFFER is used for all buffers throughout an application. That is, each DAPL buffer is implemented as an instance of BUFFER, and the DAPL buffer name is implemented as the name of a BUFFER pointer. For example, the DAPL buffer b, from above, is used to generate the following C statement.

```
BUFFER *b;
```
Chapter 7. Implementation of Distributed Applications

Where **BUFFER** is defined by:

```c
typedef struct buffer
{
    struct item *head;
    struct item *tail;
    void (*arrival)(); /* data arrival code */
    void (*pre_remove)(); /* pre remove code */
    void (*post_remove)(); /* post remove code */
} BUFFER;
```

The members of type **BUFFER** is initialised as follows. Since a buffer is a queue, the head and tail pointers within this **BUFFER** definition are pointers to the head and tail of the queue. The last three data members of **BUFFER** are function pointers which refer to the three buffer routines provided by the DAPI programmer. These routines are implemented as C functions, and to distinguish routines of different buffers in the same process, each function name generated is prefixed with the buffer’s name. For example, the three function names for a buffer called `b` are `b_data_arrival()`, `b_pre_remove()` and `b_post_remove()`.

**Creation of BUFFERS.** **BUFFER** instances are automatically created when a process commences. For each buffer, the process invokes the function called `create_buffer()` which is hidden from the user. This function returns a pointer to a new instance of the **BUFFER** structure initialised to be empty. Immediately after creating a buffer, the buffer’s function pointers are initialised to refer to the appropriate functions. For example, for a DAPI buffer called `b`, the C code generator places the following four C statements into the generated C program in order to create and completely initialise a **BUFFER** instance:

```c
b = create_buffer();
b->arrival = (void *)b_data_arrival;
b->pre_remove = (void *)b_pre_remove;
b->post_remove = (void *)b_post_remove;
```

**Structure struct item.** As mentioned above, a buffer is implemented as a queue. The following structure `struct item` indicates that the queue is implemented as a linked list, in which each node has a pointer to some buffer data which is actually one element of the buffer stored in a structure called `buffer_data`.
struct item
{
    struct buffer_data *data;
    struct item *next;
};

**Structure struct buffer_data.** This structure is used to store a buffer element, i.e., an array of data. The data array is referred to by the generic pointer called array. The number of elements in this array is stored in num_elements. The size in bytes of each array element is stored in element_size. The type of array elements is described within type_info, and if the type of array elements is a structure, then the number of data members of this structure is stored in num_type_info_elements. That is, arrays of structures can be also be sent from one process to another.

```c
struct buffer_data
{
    void *array;
    int num_elements;
    int element_size;
    struct type_info type_info[1024];
    int num_type_info_elements;
};
```

**Meta Data.** In addition to sending an array of raw data from one process to be placed into a buffer of another, associated meta data accompanies the array in transit. This meta data is stored in the structure members called type_info and num_type_info_elements of the above buffer_data structure that contains the array.

The meta data includes not only the number of elements within the array, but also the size of each element. However, it also includes data which at first appears redundant because it describes the type of each array element, whether it be a fundamental type or structure.

On occasions, this meta data about the types can be quite useful. For example, during implementation of the DAS system, this type information was used to confirm that correct data was sent and received. In other cases, this type information is absolutely essential when the data format of the sending machine differs from the data format of the receiving machine. For example, the size of an integer might be 2 bytes on the sending machine, but 4 bytes on the receiving machine. Therefore, the
incoming 2 byte integer must be converted to a 4 byte integer by the receiving process. As another example, the byte order of an integer on one machine might differ to that on another machine, and so the data bytes must be correctly reordered by the receiving process.

7.2.2 Implementation of Process Outputs

A process of a distributed application can have zero or more outputs. It is feasible that a process contains no outputs, for example, when it receives data from another process, processes this data, and writes resultant data to a file.

Every output has a buffer on the destination end. Logically, the DAPI programmer views an output and associated buffer like a conveyor belt (the output) that has a storage bin (buffer) at the destination end. The DAPI programmer need only place data on an output in order for that data to be automatically transmitted along the output, and towards the buffer. Data that reaches the buffer end of the output is automatically inserted into the buffer.

The implementation of outputs shown in Figure 7.2 must cater for the following cases:

- An output starts at one process P and ends at another process Q.
- Processes P and Q are allocated to the same machine.
- Processes P and Q are allocated to different machines.
- The number of outputs of the same type from process P to Q is many.
- The number of outputs of different types from process P to Q is many.
- There might be outputs from process P to Q, and outputs from process Q to P.

![Figure 7.2](image_url)  
Several logical communication channels between two processes are implemented using one pair of sockets to form one communication channel.
In a UNIX implementation, outputs are implemented using sockets because they support outputs that start at a process on one machine and end at a process on another machine allowing for distribution.

A socket not only supports bi-directional data flow, but also supports untyped data transmission. Therefore, only one pair of sockets is required to implement many outputs between two processes as depicted in Figure 7.2. All sockets are created using TCP [Stevens (1990) and Feit (1993)] to provide connection oriented and reliable data communications.

The DAPI programmer logically views sending data from one process to another as sending an array of like typed elements. However, the data packet that is actually sent over the socket contains not only the data array, but also a header and type description. This data packet is described in the following section (Section 7.2.3).

The DAPL programmer is provided with one function from the DAS library to send data. This function is called \texttt{DA-send()}, and is used to place an array of data onto an output. The effect of invoking \texttt{DA-send()} is that this array is placed into a new internal data packet (IDP), and this packet is sent to the destination process via a socket. In addition to the library function \texttt{DA-send()}, the DAPL programmer can provide two routines\(^1\): one which is automatically invoked immediately prior to executing \texttt{DA-send()}, and the second routine is automatically invoked immediately after executing \texttt{DA-send()}.

Outputs are automatically created and initialised at the commencement of execution of a process, and therefore, there is no need for a DAPL programmer to code the creation of outputs. The specification of an integer output, say \(x\), using the DAPI language can be as simple as:

\[
\begin{align*}
\text{output} \\
\text{int } x \\
\text{end}
\end{align*}
\]

and reference to this output is by the output's name, i.e., \(x\). For example, DAPL statements to declare an array of 3 integers, and send this array via output \(x\) can be:

\[
\begin{align*}
\text{output} \\
\text{array } x \\
\text{end}
\end{align*}
\]

\[\begin{align*}
\text{DA-send}(x) \\
\text{end}
\end{align*}\]

\(^1\) These two routines, like that associated with a buffer, are associated with an output to provide a means to specify code that must be executed immediately before or after sending data using this output, e.g., this code might check pre-conditions and post-conditions associated with sending data using a particular output.
int data[] = {123, 456, 789};

and

DA_send(x, data, 3);

However, a process might be associated with many sockets, and therefore the process invoking DA_send() must contain information to correctly select the appropriate socket to send the data. This information is stored in the data structure associated with output x during this structure’s initialisation. This information is a socket descriptor, and the data structure is based on the type called OUTPUT.

Three C data structures supporting DAPL outputs are hidden from the DAPL programmer, and are described below.

**Type OUTPUT.** A type called OUTPUT is used for all outputs throughout an application. That is, each DAPL output is implemented as an instance of OUTPUT, and the DAPL output name is implemented as the name of an OUTPUT pointer. For example, the DAPL output x, from above, is used to generate the following C statement.

```
OUTPUT *x;
```

Where OUTPUT is defined by:

```
typedef struct flow_info
{
    int sd;
    int bid;
    int installed;
    int tsize;
    struct type_info *type_info;
    int num_type_info_elements;
    void (*pre_send)();
    void (*post_send)();
} OUTPUT;
```

The members of type OUTPUT are determined as follows. As described above, sending data on an output must be sent on the appropriate socket, and as such the socket descriptor is stored within sd of an OUTPUT instance. An integer that identifies a buffer with respect to the receiving process is stored in bid (buffer ID). When sending a data array to another process, the bid value must accompany this
data array, so that the receiving process can automatically insert the array into the
correct buffer. The installed value is a boolean to indicate whether the DAPL
output is installed on the socket. Several DAPL outputs may share one socket, and if
the socket is to close, then the installed DAPL outputs that share the socket must be
uninstalled before closing the socket. In addition, data cannot be sent via an output if
it has not yet been installed. tsize represents the size, in bytes, of the output's type.
Like a buffer, type_info and num_type_info_elements are used to describe
the type. The last two data members of OUTPUT are function pointers which refer to
output routines provided by the DAPL programmer. These routines are implemented
as C functions, and to distinguish routines of different outputs within the same
process, each function name generated is prefixed with the output's name. For
example, the two function names for output x are x_pre_send() and
x_post_send().

Creation and Initialisation of OUTPUTs. OUTPUT instances are automatically
created when a process commences. For each output of a process, the process invokes
the function called create_output() which returns a pointer to a new OUTPUT
instance that is then initialised with a socket descriptor, a destination buffer ID, the
type's size and description, and function addresses for the two function pointers
pre_send and post_send.

Structure struct type_info. A description of the type of an output is stored in an
array of structures, one structure for each type described. Several such structures are
used if the output's type is a structure, one for each data member. Structure struct
type_info contains the type description which is:

```c
struct type_info
{
    int type_id;
    int struct_id;
    int type_size;
    int num_elements;
};
```

If the type being described is a fundamental type such as char, int or float, then an
identification number of this fundamental type is stored in type_id, and 0 is
stored into struct_id. The size, in bytes, of this fundamental type is stored into
type_size. The number of elements of a fundamental type is 1, so 1 is stored into
num_elements. Identification numbers associated with fundamental types are
defined in the DAS header file called type_id.h.
If the type being described is a structure, then the number of data members of this structure is stored in `num_type_info_elements` of the `OUTPUT` structure, and a type_info structure is created to describe each data member. If a data member is another structure, then its structure identification number is stored in `struct_id`, and -1 is stored in `type_id`. Identification numbers of structures are internally generated by the DAS compilation system. The size, in bytes, of the structure is stored into `type_size`, and, like other types, if this member is an array, the number of array elements is stored in `num_elements`.

**Example: Describing Fundamental Types and Structures.** The data in one type_info structure describes a fundamental type such as `char`, whereas several type_info structures are required to describe programmer defined structures in which case one type_info structure is used for each structure member. Pointers cannot be sent within this implementation and so are not considered here.

Data values used in type_info structures for describing the types `char` and `int`; a structure; and a nested structure are presented below. For each type_info structure, four values are required which are presented to the right of the type in the following examples.

**Fundamental Types.**

<table>
<thead>
<tr>
<th>type</th>
<th>type_id</th>
<th>struct_id</th>
<th>type_size</th>
<th>num_elements</th>
</tr>
</thead>
<tbody>
<tr>
<td>char</td>
<td>1</td>
<td>-1</td>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>int</td>
<td>3</td>
<td>-1</td>
<td>4</td>
<td>1</td>
</tr>
</tbody>
</table>

**Structure.**

```c
struct data
{
    char a;
    int b[10];
    float c[100];
};
```

**Nested Structures.** In this example, `struct salary` is used to define a data member of `struct employee`.

```c
struct salary
{
    float gross;
    float tax;
};
```

---

1. The sizeof() function is used to determine the value of the `type_size` member.
7.2.3 Sending Data: Internal Data Packet

Logically, the DAPL programmer views sending data from one process to another as sending an array of several elements as a single entity. The logical mechanism to send such an array is to place the array onto an output, and then the DAS system conveys this array to the destination process and inserts it into a buffer of the destination process.

This single entity can be regarded as a logical data packet (LDP). However, to send the packet requires the following additional information which is placed in the header of an internal data packet (IDP) which is defined as:

```c
struct header {
    int buffer_id;
    int num_elements;
    int element_size;
    int num_type_info_elements;
};
```

The destination buffer ID is stored in buffer_id. It is sent to the runtime system of the receiving process to identify the buffer into which the incoming data is to be inserted because several DAPL outputs, from process P to process Q, are multiplexed into one channel. The runtime system of the receiving process inserts the packet into the buffer identified by the buffer ID.

The product of the number of array elements (stored in num_elements) and the size of each element (stored in element_size) ensures that the receiving process reads the correct number of bytes from the socket to form the data array. If the array type is a structure, the header also contains the number of data members of that structure in num_type_info_elements.

**Internal Data Packet (IDP).** The logical data packet requires a header to form an internal data packet (IDP). An IDP has three parts: a header (as described above), type descriptions, and data array. The description of the array type follows the IDP header, and the actual LDP (the data array) is stored in the last part of the IDP.
So, sending a logical data packet from one process to another requires the following six activities. The sending process:
1. creates a LDP from the data array (which is available from the second argument of the DA_send() function call),
2. creates an IDP, and
3. writes the IDP on to the socket.

The receiving process:
4. reads the IDP from its socket,
5. unpacks the IDP, and
6. inserts the LDP into the appropriate buffer.

7.3 DAS System Implementation

The purpose of the distributed application system (DAS) is to manage several distributed applications and to provide a user interface called DASH for a DAS operator. The DASH interface provides commands to create or terminate application instances.

The DAS system consists of four major components:
- the DASH user interface,
- a compilation system,
- a set of data structures to contain information about application specifications and application instances, and
- a set of functions to manage applications.

In addition to these major components, the DAS system provides a known port for communications between it and application processes, and requires each process to provide a port for communications with the DAS system. Hence, during the instantiation of an application, this information is passed by messages between the DAS system and new processes.

Implementation details of these four major components and also the DAS messages are presented in the following subsections.

7.3.1 Distributed Application Shell (DASH) Implementation

An overall description of the DASH user interface is presented in Section 6.1. It is implemented as a DAS library function called dash() which is one of the first functions invoked when the DAS system process initialises itself. The first function of the DAS process sets UNIX signal dispositions, i.e., some signals are caught and handled, and others are ignored. The second function creates the main DAS data
structure. The third function processes any command line arguments. The fourth function is `dash()`. The last function deallocates the DAS data structures because the DAS system is terminating.

One of the most important features of the `dash()` function is a consequence of it being an integral part of the DAS system process. That is, the DAS system process must respond to both incoming messages from application processes, and DASH commands from the operator. Hence, the `dash()` function must be able to detect commands from stdin, and incoming messages from application processes. This is achieved by implementing the `dash()` function to block for input on both file descriptors and socket descriptors. Upon data arriving via one of these descriptors, the `dash()` function responds accordingly.

Pseudo code of the `dash()` function is presented in Figure 7.3. It shows that a prompt is printed and the function loops while the DAS process is running. At the beginning of a loop iteration, the DAS process blocks, waiting for input to arrive on any of the file or socket descriptors. If some inputs arrive they are all processed. If the DAS operator types a DASH command, then the incoming data on stdin is processed by obtaining the command string, processing this string, and then displaying another prompt. Otherwise, if the incoming data is a message from a socket, then it is obtained and processed.

dash()

PRINT "dash>>"

WHILE DAS system is running DO
    BLOCK on stdin and sockets
    FOR each ready descriptor DO
        IF stdin descriptor THEN
            get DASH command from keyboard
            evaluate DASH command
            PRINT "dash>>"
        ELSE IF socket descriptor THEN
            get message from socket
            evaluate message
        IF_END
    FOR_END
WHILE_END
END

Figure 7.3  Pseudo code of the `dash()` function.
7.3.2 DAS Compilation System Implementation

The DAS compilation system is outlined in Section 6.2. It depends on the operator loading DAL specifications and DAPL programs into the DAS system using the DASH `addapp` and `addprog` commands, and then starting the compilation system by using the DASH `compile` command.

The DAL and DAPL parsers are invoked when the operator loads the DAL specifications and DAPL programs into the DAS system. In addition to extracting (and storing) information from the specifications and programs, parsing ensures that the specifications and programs are valid and the interfaces of DAPL programs match those in the DAL specifications. This means that information about the DAL specifications and DAPL programs required by the compiler can be obtained from data structures of the DAS system.

The DAS compilation system produces object modules, one for each application process, as follows:

- The DAL specifications are first converted to canonical form, then integrated into a single DAL specification which is placed into the file called `final.dal`.
- The C code generator uses `final.dal` and the output of the DAPL parser to produce C source files, a makefile, and a createfile.
- The generated makefile is used by the make tool to invoke the actual C compiler which compiles object modules from the generated C source files.

The `compile` command calls `dash_compile()` which has two phases. In the first phase, all C files necessary for compiling the object modules are generated. However, this will fail if the DAL or DAPL parsers detect errors. If the first phase is successful, then the second phase is to compile the generated files to produce object modules using the make tool and generated makefile. Pseudo code for function `dash_compile()` is presented in Figure 7.4. Lines 3 and 4 are most important statements because they are an abstraction of much functionality for validating the DAL and DAPL specification, and generating all required files such as C source files, makefile, and createfile.

```
  1 dash_compile(application names)
  2     FOR each application name DO
  3        check DAL and DAPL specifications
  4        generate files
  5        IF parsing and file generation succeeds THEN
  6           execute make tool
  7        IF_END
  8     FOR_END
  9 END

Figure 7.4 Pseudo code of the dash_compile() function.
```
7.3.3 DAS Data Structures

While the DAS system process is running, several kinds of data need to be stored in order to manage distributed applications. The DAS system must have the following information:

1. the unique identification numbers of applications,
2. creation parameters such as the order in which to create application processes,
3. available machines to allocate application processes,
4. currently running applications such as whether all processes exist,
5. currently running applications such as port numbers and machine names in order that the DAS system can send messages to the application processes,
6. its port number and host name to which application processes can send messages to itself,
7. its socket descriptor associated with the port number, and
8. its own state.

The DAS system has one main structure in which the above information is stored. Within this structure are three linked lists: a list of machine names, a list of information about DAL specifications and DAPL programs, and a list of information about existing application instances. The main DAS system structure and the structures used to implement these lists are defined in the header file `das_structs.h`.

Creation of the DAS System Structure. After setting signal dispositions, the second task performed by the DAS process is to create the DAS system structure `struct das`. This is achieved by calling `das_create()` which returns a pointer to such a `struct`. Pseudo code of the function `das_create()` is presented in Figure 7.5.

```c

das_create()
    Allocate memory for a struct das.
    Set the DAS state to SYSTEM CREATION.
    Initialise the linked list of host name to empty.
    Initialise the linked list of DAL applications to empty.
    Initialise the linked list of applications to empty.
    Set the hostname to the host machine name.
    Set the port number to an initial value, e.g., 6000.
    Create a socket based on the port number.
    Store the socket descriptor.
    Set the last application identification number to 0.
    Change the DAS state to SYSTEM RUNNING.
    Return a pointer to the DAS system structure.

END
```

Figure 7.5 Pseudo code of the `das_create()` function which is used to create the DAS system structure.
**Structure struct das_DAL_application.** The information about DAL specifications and DAPL programs for each application is stored in a linked list that is implemented using the das_DAL_application structure. Each element of this list contains information about an application, and refers to: a structure containing information from the DAL specifications; another linked list containing information about each DAPL program associated with these DAL specifications; and a pointer to the next das_DAL_application structure.

This information extracted by the DAI and DAPL parsers and stored in these structures called struct dal_application and struct dapl_program, and defined in the header files dal_structs.h and dapl_structs.h respectively.

![Diagram](image)

*Figure 7.6* A diagram depicting the relationships among the DAS system structure, das, and structures required to store information about three applications. $ denotes a das_DAL_application structure, # denotes a dal_application structure, and + denotes a dapl_program structure.
The relationships between these structures and linked list are depicted in Figure 7.6. This figure represents an abstract view of the DAS system structure and associated linked lists for the storage of information for three applications: the first application consists of four processes; the second of two processes; and the third of three processes.

**Structure struct das_application.** After the DAS system has created an instance of a distributed application, it stores information about this new application into a structure called struct das_application. This information includes the unique identification number of the application, UNIX PID numbers of each application process, and whether or not outputs are open. All this information is described in Section 6.3.2, and is stored in the following structures which are defined in the header file das_structs.h.

- **struct das_application**
  - A linked list to store information about an application instances.
- **struct das_host**
  - A linked list of machine names.
- **struct das_process**
  - A linked list to store information about application processes, one list per application instance.
- **struct das_output**
  - A linked list to store information about process outputs, one list per process.
- **struct das_process_pre_move**
  - Information about the state of the process prior to a move operation.
- **struct das_output_pre_move**
  - A linked list to store information about all open outputs of a process, prior to the process move operation.

Figure 7.7 depicts the relationships between these structures. In particular, it depicts the DAS system structure referring to a linked list of several das_application structures (only one element of this list is shown).

From Figure 7.7, the first das_application structure refers to a linked list of four host names, and a linked list of das_process structures (only two are shown because of the page size). These host names are the ones to which all application processes are restricted to exist.

Each das_process structure corresponds to exactly one application process. The first das_process structure in Figure 7.7 corresponds to a process that is associated with two hosts and consists of four outputs. The second das_process structure corresponds to a process associated with three hosts and consists of two outputs.
7.3.4 DAS Messages

During particular operations of an application process, information must be sent to the DAS system process. For example, an application process creates a socket during initialisation to provide a communications channel between the DAS system and itself, and so the port number of this socket must be delivered to the DAS process.

The format of such a message consists of the following seven fields.

- message length,
- receiver identification number and name,
- sender identification number and name,
- a message name, and
- optional data.

There are two groups of messages associated with the DAS system. One group consists of messages sent from the DAS system to an application process. The second group contains those that are sent from application processes to the DAS system. These messages are internal to the operation of the DAS system, and are hidden from DAS operators, DAL programmers, and DAPL programmers.

Both groups of messages are listed and described in the following subsections. This is followed by descriptions, focusing on the usage of DAS messages for creating and terminating an entire application, opening and closing an output, creating and terminating an application process, and moving a process.

7.3.4.1 Messages from the DAS Process to an Application Process

Eight messages can be sent by the DAS system process to an application process. The names of these messages are prefixed with DAS to denote that the message is sent by the DAS system process. These message names and a brief description of these messages are:

1. **DAS_OPEN_DEST**  
   Open output destination.
2. **DAS_OPEN_SOURCE**  
   Open output source.
3. **DAS_CLOSE_DEST**  
   Close output destination.
4. **DAS_CLOSE_SOURCE**  
   Close output source.
5. **DAS_MOVE**  
   Instruct a process to start moving.
6. **DAS_RUN**  
   Instruct a process to start execution.
7. **DAS_SHUTDOWN**  
   Instruct a process to shut down.
8. **DAS_EXIT**  
   Instruct a process to terminate.

An application process recognises and reacts to these messages using its internal runtime system. These messages are sent by the DAS system process to a pre-established identifier (port number) associated with the application process. The DAS system process received, and stored, this identifier as part of the PROCESS_SERVER_PORT
message that is sent by the application process during its initialisation (this is described in Section 7.3.4.7 Process Creation).

7.3.4.2 Messages from an Application Process to the DAS Process

Eleven messages can be sent by an application process to the DAS system process. The names of these messages are prefixed with PROCESS to denote that the message is sent by an application process. The names and a brief description of these messages are:

1. PROCESS_SERVER_PORT  
   Incoming server port number.
2. PROCESS_DEST_PORT  
   Incoming destination port number.
3. PROCESS_SOURCE_PORT  
   Incoming source port number.
4. PROCESS_CLOSE_DEST  
   Destination of output has closed.
5. PROCESS_CLOSE_SOURCE  
   Source of output has closed.
6. PROCESS_RUNNING  
   Process is running.
7. PROCESS_SHUTDOWN  
   Instruct DAS to shut down process.
8. PROCESS_EXIT  
   Instruct DAS to terminate process.
9. PROCESS_EXITAPP  
   Instruct DAS to terminate application.
10. PROCESS_SOURCE_PROC_MISSING  
    Process terminated unexpectedly.
11. PROCESS_DEST_PROC_MISSING  
    Process terminated unexpectedly.

7.3.4.3 Application Creation

Creating and running an application requires: creating all processes, then creating all outputs, and then running the application processes. A pseudo code algorithm for creating and running an application is presented in Figure 7.8.

```
Create_Application
Create all processes of the application
WHEN all PROCESS_SERVER_PORT messages have arrived DO
   Create all outputs of the application
END_WHEN
WHEN all PROCESS_SOURCE_PORT messages have arrived DO
   Obtain the ordered sequence to start processes
   FOR each process within this sequence DO
      Send a DAS_RUN message to the process
      Wait for the PROCESS_RUNNING message
   END_FOR
END_WHEN
END
```

Figure 7.8  Pseudo code for creating and running a distributed application.
The messages required to create and run an application are those described within the sections on creating a single process (Section 7.3.4.7) and a single output (Section 7.3.4.5). However, coordination is essential because an output cannot be created until both the source process and destination process have been created, and the application cannot commence until all outputs have been created. This coordination is controlled by the following four messages:

- a particular PROCESS_SERVER_PORT message,
- a particular PROCESS_SOURCE_PORT message,
- the DAS_RUN message, and
- the PROCESS_RUNNING message.

While creating all processes, each new process sends one PROCESS_SERVER_PORT message to the DAS system which indicates that the process has been created. However, during application creation, the PROCESS_SERVER_PORT message of interest to the DAS system is the last one because it signifies that all processes of this application have been created and are waiting to create outputs.

Four messages are used to create an output. The last of these four messages, PROCESS_SOURCE_PORT, signifies to the DAS system that the creation of the output has completed. However, there might be many outputs in a particular application, and therefore, the last PROCESS_SOURCE_PORT message received during application creation signifies to the DAS system that all outputs are created. After creating all outputs, the DAS system sends a DAS_RUN message to each process to instruct it to commence executing their DAPL program.

DAS_RUN messages are sent in the order specified within the creation clause of the DAL specification. However, after sending a DAS_RUN message the DAS system waits for a PROCESS_RUNNING message from the process before sending the a DAS_RUN message to the next process. This ensures that processes are started in a controlled manner.

### 7.3.4.4 Application Termination

The termination of a distributed application requires all application outputs to be closed, and then all application processes to be terminated. Given that all outputs connected to a single process are closed prior to the process terminating, then terminating an application is simply implemented as terminating each application process, one after the other. A pseudo code algorithm for terminating an application is presented in Figure 7.9.
Terminating an application commences with sending the DAS_SHUTDOWN message to one of the application’s processes. This causes all outputs connected to this process to be closed. Just prior to a process terminating, it sends a PROCESS_EXIT message to the DAS system to instruct the DAS system to send another DAS_SHUTDOWN message to the next process of the terminating application. This continues until all processes have terminated. In addition, the DAS system deletes all information about this application instance from its data structures after the last PROCESS_EXIT message arrives.

7.3.4.5 Output Creation

An output is required from application process P to process Q in order to send data from P to Q. The creation of such an output is dependent on four messages. Two are required to create the destination end of the output in the receiving process Q, the other two messages are required to create the source end in the sending process P.

These four messages and the sequence in which they occur are:

1. DAS_OPEN_DEST
2. PROCESS_DEST_PORT
3. DAS_OPEN_SOURCE
4. PROCESS_SOURCE_PORT

Figure 7.10 depicts the sequence of these four messages for creating an output with respect to the DAS system, the sending process, and the receiving process.

The first message, DAS_OPEN_DEST, is sent by the DAS system process to the receiving process to instruct it to create the destination end of the output. If a socket does not exist for this output, then one is created.

The DAS system process sends the first message, DAS_OPEN_DEST, either as a result of: the DAS system process creating an application and is in the phase of creating outputs; the DAS operator using the open DASH command to manually create an output; or the DAS operator using the createproc DASH command to create a process of an existing application manually and so outputs associated with this new process are automatically created.
After creating the destination end of the output, the receiving process sends the second message, `PROCESS_DEST_PORT`, and the port number of the socket to the DAS system. After sending the port number, the receiving process blocks, waiting for the sending process to complete the connection.

The sending process can only connect to the receiving process if it has two values: the port number that was sent to the DAS system, and the name of the host on which the receiving process exists. The DAS system has just received the port number and has access to the host name, and sends both values to the sending process with the third message, `DAS_OPEN_SOURCE`, so that the sending process creates the source end of the output.
The fourth message, `PROCESS_SOURCE_PORT`, is sent by the sending process to the DAS system. This informs the DAS system that the source end of the output is established. This message also contains the port number which the DAS system stores into the `das_output` structure associated with this new output.

### 7.3.4.6 Output Termination

The termination, or closing, of an output is the removal of the output from the application. This is achieved by closing both ends of the output. There are two situations in which an output is closed: one of the two processes attached to the output terminates, e.g., during application termination, or process moving; the second situation is when the DAS operator uses the DASH command `close` to purposely close an output.

The termination of an output is handled by the following four messages:

1. `DAS_CLOSE_DEST`
2. `PROCESS_CLOSE_DEST`
3. `DAS_CLOSE_SOURCE`
4. `PROCESS_CLOSE_SOURCE`

Figure 7.11 depicts the sequence of four messages, and three processes, involved in terminating an output. Like opening an output, two messages are associated with the output’s destination end, the other two are associated with the output’s source end.

The first message, `DAS_CLOSE_DEST`, is sent by the DAS system to instruct the receiving process to close the destination end of the output. In addition, if this output is the last one to exist on the supporting socket, then the receiving process closes the socket as there is no need to keep the socket open.

The second message, `PROCESS_CLOSE_DEST`, is sent by the receiving process to inform the DAS system that the destination end of the output is closed, and the DAS system updates its database to reflect this change. Thereafter the third message, `DAS_CLOSE_SOURCE`, is sent by the DAS system to the sending process.

The third message instructs the sending process to close the source end of the output. Like the receiving process, the sending process also closes the socket when not in use.

The fourth message, `PROCESS_CLOSE_SOURCE`, is sent by the sending process to inform the DAS system that the source end of the output is closed, whereby the DAS system updates its data to reflect the closure of this output.
7.3.4.7 Process Creation

The creation of a distributed application requires the creation of all processes of an application on computers of a network, followed by the opening of sockets, as presented in Section 7.3.4.5 Output Creation, to create application outputs. Creating an application of several processes is based on creating one process after the other and is presented in this section.

The creation of one application process is dependent on the connectivity of this process within the distributed application, and the creation of the entire application consists of the following three stages:
1. Partially create all application processes. That is, create all the UNIX processes, but force them to block while all outputs are formed.
2. Create all application outputs and necessary sockets.
3. Start the distributed application. That is, complete the creation of application processes by unblocking them so that they execute with respect to the application’s functionality.

The following three messages are used to create an application process (ignoring messages associated with creating an output). Figure 7.12 depicts these three messages being sent from (to) the application process to (from) the DAS system.

1. PROCESS_SERVER_PORT
2. DAS_RUN
3. PROCESS_RUNNING

The first message, PROCESS_SERVER_PORT, is sent by the application process to the DAS system. This message signifies that process initialisation has completed, and that the process is blocked and ready to be connected to other application processes. This initialisation phase includes:

- creating a socket,
- sending the port number\(^1\) associated with this new socket as part of the PROCESS_SERVER_PORT message, and
- blocking while all outputs of the application are created.

The second message, DAS_RUN, sent by the DAS system to the application process, means that the connectivity of the application is complete, and that the process can unblock and start execution.

The third message, PROCESS_RUNNING, is sent to the DAS system by the application process as confirmation that this process is running.

7.3.4.8 Process Termination

The termination of an application process can occur for the following reasons:

- The entire application terminates, implying that all application processes terminate. This can occur when a process invokes the DAS library function \texttt{DA\_exitapp()}, or the operator invokes the DASH command \texttt{exitapp}.
- An application is directly terminated by itself or the DAS operator. The process invokes the DAS library function \texttt{DA\_exit()} to terminate itself. In the second case, the DAS operator uses the DASH command \texttt{exitproc}.

In either case, outputs that enter and leave the terminating process must be closed prior to process termination.

\(^1\) This port number allows the DAS system to connect to and communicate with this application process.
The sequence of messages required to create an application process.

The termination of an application process is handled by the following four messages (ignoring messages associated with closing outputs). Figure 7.13 depicts the sequence of these messages with respect the DAS operator terminating a process using the `exitproc` command.
1. **DAS_SHUTDOWN**
2. **PROCESS_SHUTDOWN**
3. **DAS_EXIT**
4. **PROCESS_EXIT**

![Diagram](image)

**Figure 7.13** The sequence of messages required to terminate an application process.

The first message, **DAS_SHUTDOWN**, is sent by the DAS system to the application process. This message causes the receiving process to change state to **terminating**, invoke the **DA_exit()** user library function, and send the **PROCESS_SHUTDOWN** message to instruct the DAS system to close all outputs connected to the terminating process. The **DA_exit()** function cannot directly terminate the process, say, by using the UNIX **exit()** function, because the outputs have not been closed.
The third message, DAS_EXIT, used by the DAS system to initiate the final stages of termination, is sent to the terminating process to invoke the DAS function _DA_exit() which:

- sends the PROCESS_EXIT message, along with the exit status of the process, to the DAS system,
- closes the socket to which the DAS system communicates with this process, and
- invokes the UNIX exit() function along with the exit status.

The fourth message is called PROCESS_EXIT. Its purpose is to not only inform the DAS system that the application process has terminated, and therefore, the DAS system can update its data base regarding the process, but also provide an exit status value to the DAS system (akin to a UNIX process providing an exit status value to the operating system on exit).

7.3.4.9 Process Moving
Moving an application process is described in terms of processes, buffers and outputs in Section 6.1.3. Moving a process from one machine to another consists of three stages:

- saving state information about the process,
- terminating the process, and
- creating a new process on another machine.

The move operation ensures that the connectivity of the process before the move is the same as after the move. For example, if a process of three outputs has only one output open before the move, this output should be the only one open after the completion of the move. However, the move operation does not take care of open resources such as files. it is the responsibility of the designer to ensure that resources are closed before a move, and appropriate resources are opened after the move.

Moving an application process from its current machine to another machine is initiated by the DAS operator using the DASH move command which causes the DAS system to store information about the process, and then send the DAS_MOVE message to the process. The following messages are sent during a move operation:

1. DAS_MOVE
2. PROCESS_SHUTDOWN
3. DAS_EXIT
4. PROCESS_EXIT
5. PROCESS_SERVER_PORT
6. DAS_RUN
7. PROCESS_RUNNING

Figure 7.14 depicts the sequence of these messages with respect the DAS operator moving a process using the move command.
Figure 7.14 The sequence of messages required during a process move operation.
These messages are those used for process termination and creation, except that the first message, DAS_MOVE, replaces the first message, DAS_SHUTDOWN, in termination sequence. Moving a process requires four additional tasks than for process termination and creation. These are:

1. The DAS system must save process information immediately before sending the DAS_MOVE message.
2. The process invokes a pre_move function immediately after receiving the DAS_MOVE message.
3. The DAS system opens the correct number of outputs (as were open before the move) immediately before sending the DAS_RUN message.
4. The new process invokes a post_move function immediately after receiving the DAS_RUN message.

7.4 Conclusion

The implementation of the DAS run-time system, its internal data structures and messages, DASH user interface, the new DAL and DAPL programming languages, DAI and DAPL parsers, and the compilation system has been quite successful. The DAL and DAPL languages allow a programming to produce distributed applications using few statements that are easy to understand.
8 Testing the DAS System and Distributed Applications

Testing the DAS system was performed during and after development. Testing during development focused on unit and integration testing to verify that components of the DAS system worked correctly and successfully created the processes and sockets required to support a distributed application.

After development, testing focused on system level behaviour such as verifying that the DAS system correctly compiled DAL specifications and DAAL programs. That is, various distributed applications were specified in DAL and DAAL, compiled and executed to determine if the DAS system could be faulted to reveal errors.

Different applications were developed for system level testing of the DAS system in order to demonstrate that the research goals presented in Section 1.2 and Section 1.3 were achieved. The tasks to achieve these goals are presented in Section 8.1, and the applications used in testing and test results are presented in Section 8.2, Section 8.3 and Section 8.4.

Although an unlimited number of distributed applications can be built with the DAS system, there is a limited numbers of fundamental ways in which processes of an application can be connected. For example, the simplest combination is two processes that are connected by one output. Even using this simple combination, an arbitrary number of applications can be formed by using an assembly line arrangement as in Figure 8.1.

![Figure 8.1](image)

*Figure 8.1* A distributed application consisting of N processes connected in an assembly line fashion.

**Fundamental Arrangements.** The number of fundamental connectivity arrangements is nine (depicted in Figure 8.2 to Figure 8.10). These arrangements depend on:

- the number of outputs connected to the same destination process,
- the number of buffers within the same destination process,
• the orientation of outputs between two processes,
• the type of data that is sent from one process to another, and
• the number of elements sent from one process to another.

**Fundamental Hierarchies.** In addition to testing several fundamental connectivity arrangements of distributed applications, testing of hierarchical DAL specifications is required. Again, there is an unlimited number of such hierarchies because a particular hierarchy can be simply extended by incorporating another leaf node into the hierarchy.

There are four fundamental hierarchical DAL specifications (depicted in Figure 5.5, Figure 8.15, Figure 8.16 and Figure 8.17) consisting of the following relationships:
• parent/child,
• parent/child/grandchild,
• parent/several children, and
• several children, but the same DAL specification for each child.

For each of these 13 fundamental arrangements and hierarchies, a distributed application was specified using DAL and DAPL, compiled and executed in order to test the DAS system. Execution results were compared to expected results. The test applications and results for the 9 fundamental arrangements are presented in Section 8.2, and those for the 4 fundamental hierarchies are presented in Section 8.3. The DAL specifications, DAPL programs, and application outputs for connectivity tests and hierarchy tests are listed in Appendix G and Appendix H respectively.

### 8.1 Tasks to Achieve Research Aims

The tasks to be solved to achieve the aims of this research are:

1. to specify a distributed application using two independent specifications: one for connectivity and the other for functionality. This task is part of all tests because all application specifications have a DAL specification for connectivity and DAPL programs for functionality. This task has been successful.

2. to specify connectivity in a modular fashion to support re-use of existing connectivity specifications. This task has been achieved when creating a test application whereby one existing DAL specification is re-used three times to form the complete specification of connectivity.

3. to combine connectivity and functionality specifications to form a complete specification of a distributed application. This task is part of all tests because all applications are combined by the DAS operator using the DASH addprog command whereby the operator specifies which DAPL program is added to which DAL specification. This task has been successful.
4. to develop several different applications by using the same specification of connectivity for each, but with different specifications of functionality. This task has been achieved when creating the test applications presented in Section 8.2.1 and Section 8.2.7. Both applications have the same structure, but the functionality in the DAPL programs differ (Section H.1.2 and Section H.7.2).

5. to generate low level code from DAL specifications and DAPL programs. This task is required for all of the following tests, and has been achieved. The generated code is C code, an example of such code is presented in Appendix J.

6. to compile object modules from DAL specifications and DAPL programs. Again, this task is required for all of the following tests, and has been achieved. The DAS compilation system generates C code from DAL specifications and DAPL programs, then invokes a C compiler to produce the object modules.

7. to distribute application processes onto computers of a network by the DAS system. Process distribution is part of all tests. All application processes of all tests were automatically distributed onto specified computers. For example, process P and Q of test 1 are specified to run on computers called smaug and eldacar respectively. The output of this test (Section H.1.3) shows that process P is running on the host called smaug, and Q is on the host called eldacar; and confirms that the distribution of these processes was achieved.

8. to implement message passing between application processes over the network. Message passing was incorporated into all tests by ensuring that application processes ran on different computers. So, all test applications depended on message passing to function correctly. Message passing was achieved because each test application produced correct results.

9. to manage several applications of differing sort by creating, concurrently running and terminating these applications using the DAS system. Managing multiple applications was achieved.

Testing these tasks is presented in the following sections. Section 8.2 discusses testing fundamental connectivity. Section 8.3 discussed testing fundamental hierarchies. Testing the management of running several applications is presented in Section 8.4.

### 8.2 Testing Fundamental Connectivity Arrangements

Six of the nine test applications of fundamental arrangements have one common characteristic, namely that the data sent from one process to another are of the same type (integers were used for these six tests). The remaining three test applications are concerned with sending arrays of integers, arrays of characters (strings), and arrays of structures. A summary of these 9 tests are shown in Table 8.1.
## Table 8.1 Summary of testing the 9 fundamental connectivity arrangements.

<table>
<thead>
<tr>
<th>Test Number</th>
<th>Brief Test Description</th>
<th>Data Transferred</th>
<th>Successful Send</th>
<th>Successful Receive</th>
<th>Application Terminated</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Single output from one process to a single buffer.</td>
<td>123</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>2</td>
<td>Two outputs from one process to a single buffer.</td>
<td>123 and 456</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>3</td>
<td>Two outputs from two process to a single buffer.</td>
<td>123 and 456</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>4</td>
<td>Two outputs for bidirectional data transfer between two processes.</td>
<td>123 and 124</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>5</td>
<td>Two outputs from one process to two buffers in the same process.</td>
<td>123 and 456</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>6</td>
<td>Two outputs from two processes to two buffers in the same process.</td>
<td>123 and 456</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>7</td>
<td>Sending arrays of integers.</td>
<td>Arrays of 10, 5 and 100 integers.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>8</td>
<td>Sending arrays of integers and strings.</td>
<td>3 integer arrays, and 4 strings.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
<tr>
<td>9</td>
<td>Sending arrays of structures.</td>
<td>Arrays of 3, 2 and 4 structures.</td>
<td>Yes</td>
<td>Yes</td>
<td>Yes</td>
</tr>
</tbody>
</table>
8.2.1 Test 1: Single Output from One Process to Single Buffer

Test 1 consists of an application of two processes connected by one output (Figure 8.2). Therefore there must be a buffer at the destination end of the output. A single integer, say 123, is sent from the source process via the output to the destination process. On arrival of the integer, the destination process removes the integer from its buffer, displays the value, deallocates the buffer element, and then forces the application to terminate. In addition, the source process runs on one machine, and the destination process runs on another. These machines are specified in the DAL specification.

![Figure 8.2](image-url) An application for sending an integer from one process to another.

**Test 1 Results.** A successful transmission of the integer 123 from process \( P \) to process \( Q \) occurred. Process \( Q \) received and displayed this integer.

The expected number of processes required to create this application is two: one is specified to run on \( \text{smaug} \), the other is to run on \( \text{eldacar} \). The expected output is the integer value 123. The application is expected to terminate.

The actual results are as follows. The integer 123 was sent and displayed by process \( Q \). Information about all application processes was displayed by using the DASH status command which showed that this application consisted of two processes, \( P \) and \( Q \), such that process \( P \) is running on \( \text{smaug} \), and process \( Q \) is running on \( \text{eldacar} \). In addition, the application output (listed in Appendix H, Section H.1.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

8.2.2 Test 2: Two Outputs from One Process to a Single Buffer

Test 2 is like Test 1, but it consists of two outputs, and both outputs connect to the same buffer (Figure 8.3). For Test 2, two integers, say 123 and 456, are sent from the source process to the destination process, one integer on each output. On arrival of either integer, the destination process removes the integer from its buffer, displays the integer value, and deallocates the buffer element. The application is terminated by the destination process after both integers are displayed.
Test 2 Results. A successful transmission of both integers, 123 and 456, from process \( P \) to process \( Q \) occurred. Process \( Q \) received both integers in the same buffer, displayed both integers, and then terminated the application.

The expected number of processes required to create this application is two: one is specified to run on \texttt{smaug}, the other is to run on \texttt{eldacar}. The expected output is the integer values 123 and 456.

The actual results are as follows. The integers, 123 and 456, were sent by process \( P \) and displayed by process \( Q \). The DASH \texttt{status} command showed that this application consisted of two processes, \( P \) and \( Q \), such that process \( P \) is running on \texttt{smaug}, and process \( Q \) is running on \texttt{eldacar}. In addition, the application output (listed in Appendix H, Section H.2.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

8.2.3 Test 3: Two Outputs from Two Processes to a Single Buffer

Test 3 is like Test 2, but both outputs have their own source process, instead of the same source process as in Test 2 (Figure 8.4). However, this application consists of three processes and each run on a different machine. For this test, two integers are sent to the same destination buffer. One integer, say 123, is sent from one source process. The second integer, say 456, is sent from a second source process. On arrival of either integer, the destination process removes the integer from its buffer, displays the integer value, and deallocates the buffer element. This application is terminated by the destination process after both integers are displayed.

Test 3 Results. A successful transmission of both integers occurred. Integer 123 was sent by process \( P \) to process \( Q_1 \), and integer 456 was sent by process \( R \) to process \( Q_2 \). Process \( Q \) received both integers in the same buffer, displayed both integers, and then terminated the application.
The expected number of processes required to create this application is three: process $P$ is specified to run on smaug, process $Q$ is to run on eldcar, and $R$ runs on luthin. The expected output from process $Q$ is the integer values 123 and 456.

The actual results are as follows. The integers, 123 and 456, were sent by processes $P$ and $R$ and displayed by process $Q$. The DASHI status command showed that this application consisted of three processes: $P$, $Q$ and $R$, such that process $P$ is running on smaug, process $Q$ is running on eldcar, and process $R$ on luthin. In addition, the application output (listed in Appendix H, Section II.3.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

### 8.2.4 Test 4: Two Outputs for Bidirectional Data Transfer

The distributed application for test 4 consists of two processes connected by two outputs. Unlike the previous tests in that only one destination process was incorporated into the application, this test includes two processes, and both processes are destination processes (Figure 8.5).

The purpose of this test is to confirm that bidirectional data flow between two processes is successful. In particular, an integer, say 123, is sent from one source process via its output to a second process. On arrival of this integer, the second process removes the integer from its buffer, displays the integer value, increments the
integer and sends it to the first process, and deallocates the buffer element. When the first process receives an integer, it displays the integer value and terminates the application. Both processes run on different machines, i.e., smaug and eldacar.

**Test 4 Results.** Bidirectional data flow between two processes is successful. Integer 123 was sent by process P to process Q. Process Q incremented the integer value to 124 and sent it back to process P. Process P terminated the application after receiving and displaying the integer 124.

The expected number of processes required to create this application is two: process P is specified to run on smaug, and process Q is specified to run on eldacar. The expected output from process Q is the integer value 123 from process Q, followed by 124 from process P.

The actual results are as follows. The integer 123 was sent by process P to process Q and displayed by process Q, it was incremented by 1 and sent to process P which displayed 124. The DASHI status command showed that this application consisted of two processes: P and Q, such that process P is running on smaug, and process Q is on eldacar. In addition, the application output (listed in Appendix H, Section H.4.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

### 8.2.5 Test 5: Two Outputs from One Process to Two Buffers

Test 5 is like Test 2. That is, two processes are connected by two outputs, both outputs start at the same process and end at a second process. In the case for test 2, both outputs are connected to the same buffer within the destination process. However, for this test, test 5, both outputs are connected to distinct buffers within the destination process (Figure 8.6).

![Figure 8.6](image)

**Figure 8.6** An application to send integers from one process to two buffers within another process.

The purpose of this test is to check whether data that is sent to a process consisting more than one buffer is inserted into the correct buffer. Only one process is used to send data during this test.
The distributed application Test_5 sends two integers, say 123 and 456, from the source process to the destination process, one integer on each output. On arrival of either integer, the destination process removes the integer from the appropriate buffer, displays the buffer name and integer value, and deallocates the buffer element. The application is terminated by the destination process after both integers are displayed.

**Test 5 Results.** Sending data from one process to two distinct buffers of another process is successful. Two integers, 123 and 456, were sent by process P to process Q. Integer 123 and 456 were successfully inserted into buffers buf1 and buf2 respectively. Process Q terminated the application after receiving and displaying both integers.

The expected number of processes required to create this application is two: process P is specified to run on smaug, and process Q is specified to run on eldcar. The expected output from process Q is a pair of integers: 123 and 456, and the buffer names in which these integers were inserted.

The actual results are as follows. Process Q, on receiving either integer in one of its buffers, removed the integer from the buffer, and displayed the buffer's name and integer value. The DASH status command showed that this application consisted of two processes: P and Q, such that process P is running on smaug, and process Q is on eldcar. In addition, the application output (listed in Appendix II, Section II.5.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

**8.2.6 Test 6: Two Outputs from Two Processes to Two Buffers**

Test 6 is very similar to Test 5. They differ in that test 6 uses two processes to send data to two buffers within a third process (Figure 8.7). The purpose of this application is to confirm that data can be sent from more than one process and this data is inserted into more than one buffer in a single receiving process.

![Figure 8.7](image)

An application to send integers from two processes to two buffers within another process.
Test 6 Results. Sending data from two processes to two distinct buffers in a third process is successful. Process P sent the value 123 to buffer buf1, and process R sent the value 456 to buffer buf2. Both buffers are part of process Q. Process Q terminated the application after receiving and displaying both integers.

The expected number of processes required to create this application is three: process P is specified to run on smaug, process Q is specified to run on eldacar, and process R is to run on luthin. The expected output from process Q is a pair of integers: 123 and 456, and the buffer names in which these integers were inserted.

The actual results are as follows. Process Q, on receiving either integer in one of its buffers, removed the integer from the appropriate buffer, and displayed the buffer's name and integer value. The DASH status command showed that this application consisted of three processes: P, Q and R, such that process P is actually running on smaug, process Q is on eldacar, and process R is on luthin. In addition, the application output (listed in Appendix H, Section H.6.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

8.2.7 Test 7: Sending Arrays of Integers

Test 7 consists of an application of two processes connected by an integer output (Figure 8.8). By default, outputs of a certain type can be used to transmit an array of data of the same type. The DAS library function DA_send() is used to transmit this array by writing a DAS internal data packet (IDP) onto a socket. The size of the array is program determined and is one of the parameters of the DA_send() function.

![Figure 8.8 An application to send arrays of integers.](image)

For example, given an integer output \( x \) from process P to process Q, then it is possible to use the library function DA_send() several times from within process P to transmit several integer arrays of differing sizes to process Q. The following statements are sufficient to send three integer arrays. The first consists of the first 10 elements of the array called data. The second array consists of 5 elements, from data[10] to data[14] inclusive. The entire array is sent on the third invocation of DA_send().
int data[100];

/* initialise data array */

DA_send(x, data, 10);
DA_send(x, &data[10], 5);
DA_send(x, data, 100);

The application depicted in Figure 8.8 can be used to send three such arrays from process P to Q. The array of data and the DA_send() function calls are coded as part of the DAPL program associated with the sending process. Process Q will display the contents of each array received, and terminates the application after displaying three arrays.

Test 7 Results. Sending several integer arrays of differing sizes from one process to another via the same output is successful. Process P consecutively sent three arrays to process Q. The first array contained 10 integers, the second contained 5 integers, and 100 integers were sent as the third array. Process Q displayed, in turn, the contents of these arrays on their arrival, and terminated the application after displaying the third array.

The expected number of processes required to create this application is two: process P is specified to run on smaug, and process Q runs on eldacar. The expected output from process Q is three arrays of integers (the value of each array element is the square of the element’s index).

The actual results are as follows. Process Q, on receiving an array, removed the array from the buffer, and displayed its contents. The DASH status command showed that this application consisted of two processes, P and Q, such that process P is running on smaug, and process Q on eldacar. In addition, the application output (listed in Appendix H, Section H.7.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

8.2.8 Test 8: Sending Arrays of Integers and Strings
Test 8 is like test 7, but sends arrays of more than one type. Three integer arrays and four strings will be sent from one process to another. The application consists of two process and two outputs from one process to the other. The type of one output is int, and char is the type of the second (Figure 8.9).
Process $P$ sends the following arrays of strings and integers (in the order shown). Process $Q$ displays the contents of these arrays as soon as they arrive, and terminates the application when it receives the string END.

\begin{verbatim}
Distributed Application System (DAS)
1 2 3 4 5
DAL Language
123 456 789
DAPL Language
9 8 7 6 5 4 3 2 1 0
END
\end{verbatim}

**Test 8 Results.** Sending arrays of different types from one process to another via one underlying communication channel supporting two outputs is successful. Process $P$ alternatively sent strings and integer arrays to process $Q$ which displayed the contents of these arrays, and terminated the application when a string contained the value END.

The expected number of processes required to create this application is two: process $P$ is specified to run on smaug, and process $Q$ runs on eldacar. The expected output from process $Q$ is seven arrays, four strings and three integer arrays.

The actual results are as follows. Process $Q$, on receiving an array, removed the array from the appropriate buffer, and displayed the array's contents. The DASH status command showed that this application consisted of two processes, $P$ and $Q$, such that process $P$ is running on smaug, and process $Q$ on eldacar. In addition, the application output (listed in Appendix H, Section II.8.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

### 8.2.9 Test 9: Sending Arrays of Structures

Test 9 is like test 7. The application for test 7 sent arrays of integers from one process to another. For test 9, the application sends arrays of structures from one process to another (Figure 8.10). Three arrays are sent: the first contains three structures, the
second contains two structures, and the third has four structures. This test is to confirm that arrays of structures can be sent from one process and received by another process.

![Diagram](image)

**Figure 8.10** An application to send several arrays structures.

The structure definition used within this application consists of an integer identification number, and a string of ten characters. This structure is:

```c
struct test
{
    int id;
    char name[10];
};
```

**Test 9 Results.** Sending arrays of structures from one process to another is successful. Process P sent three arrays of structures: the first contained 3 structures, the second contained 2, and the third consisted of 4 structures. Process Q displayed the contents of these arrays, and terminated the application after displaying the contents of the third array.

The expected number of processes required to create this application is two: process P is specified to run on smaug, and process Q runs on eldacar. The expected output from process Q is the contents of three arrays. Output associated with the first array is expected to contain data of three structures such as:

1. Bob
2. Kim
3. Sue

Output associated with the second array is expected to contain data of two structures such as:

4. Angela
5. Peter
Output associated with the third array is expected to contain data of four structures such as:

6, Jane
7, Paul
8, Alan
9, Amy

The actual results are as follows. Process Q, on receiving an array of structures, removed the array from the appropriate buffer, and displayed the array's contents. The DASH status command showed that this application consisted of two processes, P and Q, such that process P is running on smaug, and process Q on eldacar. In addition, the application output (listed in Appendix II, Section H.9.3) also indicates that the application terminated successfully.

The expected and actual results match each other, confirming a successful test.

8.3 Testing Fundamental Hierarchies

A hierarchical DAL specification is a combination of DAL specifications that form a tree structure. Although an unlimited number of such trees is possible, these trees are all formed from the following small number of characteristics:

- Each child node has a parent, except for the root node.
- A child node can be a parent of another node.
- A parent node can have several children.
- The specification for each of several children can be the same DAL specification, i.e., a single DAL specification can be used several times within the same hierarchical DAL specification.

These four characteristics have been tested by producing and testing four hierarchical DAL specifications. The connectivity of four such hierarchies is depicted in Figure 8.11 to Figure 8.14. These figures are abstract in that inscriptions and buffers are not shown, but the relationships between parent and children nodes are shown are dashed lines. A summary of these tests is shown in Table 8.2. The DAL specifications, DAPL programs, and application outputs for the following four tests are listed in Appendix H.
Figure 8.11  A parent/child relationship between two DAL specifications.

Figure 8.12  A child/grandchild relationship between two DAL specifications.

Figure 8.13  Relationships between a parent and its two children.
8.3.1 Test 1: Parent and Child Relationship

The purpose of test 1 is to confirm that a distributed application can not only be specified using the smallest hierarchy, i.e., a tree of two nodes, but that this specification can be used by the DAS compilation system to produce object modules. The smallest hierarchical DAL specification consists of the root node and one child of the root. Therefore, this hierarchical DAL specification consists of two DAL specifications: one for the root node, and another for the child.

The number of processes, however, constituting such an application is not bounded. The number of processes within the DAL specification associated with the root node, and the number within the child’s DAL specification can be unlimited. But the total number of processes required to instantiate the application is the sum of processes specified within both DAL specifications, less one. One process of the parent specification is not counted because its functionality is that of the child specification, and therefore it is replaced by all processes associated with the child.
Chapter 8. Testing the DAS System and Distributed Applications

A hierarchical DAL specification sufficient for test 1 is presented in Chapter 5 and is the specification of the distributed application called Fibonacci_sequences (Figure 5.5) which computes and displays several Fibonacci sequences.

This application is specified to compute and display the following three sequences of Fibonacci numbers.

\[
\begin{align*}
1 & \quad 1 & \quad 2 & \quad 3 & \quad 5 \\
1 & \quad 1 & \quad 2 & \quad 3 & \quad 5 & \quad 8 & \quad 13 \\
1 & \quad 1 & \quad 2 & \quad 3 & \quad 5 & \quad 8 & \quad 13 & \quad 21 & \quad 34 & \quad 55
\end{align*}
\]

Process display_sequence (Figure 5.5) sends the length of a sequence to process create_sequence, the Fibonacci sequence is computed and stored into an array which is sent by create_sequence to display_sequence to display the sequence.

**Test 1 Results.** The expected number of processes required to instantiate this application is three (process display_sequence, process create_sequence and process compute Fibonacci), and the expected output is three Fibonacci sequences.

The actual number of processes created is three. This is confirmed by invoking the DASH status command to display information about the application. The actual output is the same as the expected output. In addition, the application output (listed in Appendix I, Section I.1.3) shows that the application consisted of the following three processes, and that it terminated successfully:

- display_sequence running on smaug, UNIX PID value equals 23485,
- create_sequence running on smaug, UNIX PID value equals 23479, and
- compute_Fibonacci running on luthin, UNIX PID value equals 7188.

The expected and actual results match each other, confirming a successful test.

**8.3.2 Test 2: Parent, Child and Grandchild Relationships**

Test 2 is an extension of test 1. A third node is incorporated to form a hierarchy of three nodes: a root node, a child of the root, and a grandchild of the root (Figure 8.15). The child and grandchild are from test 1. Test 2 is designed to confirm that more than one generation of DAL specifications can be used to compile the required object modules, and to check that the distributed application can be successfully created and executed.
Briefly, test 1 created and displayed a fixed number of Fibonacci sequences of differing lengths, these lengths were hard coded within this application. The distributed application of test 2 is like the Fibonacci application used for test 1. However, instead of hard-coding the lengths within the processes associated with creating the sequences, these processes have been slightly rewritten to receive an array of lengths from another process. The process that sends this array of lengths is part of the root node for test 2. Therefore, the functionality of test 2 is such that one process sends an array of sequence lengths to a second process. For each length within this array, this second process sends the length to a third process to compute a Fibonacci sequence of this length. The sequence is sent back to the second process which displays this sequence.
**Test 2 Results.** The expected number of processes required to instantiate this application is four, and the expected output is three Fibonacci sequences, the same sequences as in the previous test.

The actual number of processes created is four. This is confirmed by invoking the DASH status command. The actual application output, i.e., the three Fibonacci sequences (listed in Appendix I, Section 1.2.3), is the same as the expected output. In addition, the application output shows that the application consisted of the following four processes, and that it terminated successfully:

- `sequence_length running on smaug`, UNIX PID value equals 23606,
- `display_sequence running on smaug`, UNIX PID value equals 23584,
- `create_sequence running on smaug`, UNIX PID value equals 23583, and
- `compute_Fibonacci running on luthin`, UNIX PID value equals 7250.

The expected and actual results match each other, confirming a successful test.

### 8.3.3 Test 3: Parent of Several Children

The functionality of the distributed application for test 3, like that of test 1 and 2, is to compute and display several Fibonacci sequences. However, the difference between test 3 and tests 1 and 2 is that the hierarchical DAL specification consists of sibling nodes.

The purpose of test 3 is to confirm that a specification consisting of a parent node of several children can be used by the DAS compilation system to produce the application’s object modules, and that these object modules execute as expected.

The hierarchical DAL specification for test 3 is depicted in Figure 8.16. The root node can be used to describe this application. One process sends the length, $N$, of a Fibonacci sequence. The receiving process computes the Fibonacci sequence of length $N$ and sends this sequence back to the first process which displays the sequence and then sends the length of another sequence to the second process.

However, these two processes of the root do not exist when the application is created because their functionality is specified by two other DAL specifications, one called `sequences` that sends the lengths of sequences and displays sequences, the other called `Fibonacci` to compute a Fibonacci sequence and send it as an integer array to another process.

**Test 3 Results.** The expected number of processes required to instantiate this application is four, these are the processes within the leaf nodes of the hierarchy presented in Figure 8.16. The expected output is three Fibonacci sequences, the same sequences as produced by tests 1 and 2.
Figure 8.16 An application to display sequences of Fibonacci numbers.

The actual number of processes created is four. This is confirmed by invoking the DASH status command. The application displayed the same sequences as expected (Appendix I, Section 1.3.3). In addition, the application output shows that the application consisted of the following four processes, and that it terminated successfully:

- `sequence_length` running on smaug, UNIX PID value equals 23799,
- `display_sequence` running on smaug, UNIX PID value equals 23800,
- `create_sequence` running on smaug, UNIX PID value equals 23798, and
- `compute_Fibonacci` running on luthin, UNIX PID value equals 7298.

The expected and actual results match each other, confirming a successful test.

8.3.4 Test 4: Single DAL Specification for Several Children

The output produced by the distributed application for test 4 (Figure 8.17), like that of the previous three tests, is several Fibonacci sequences. However, the functionality for test 4 differs to that of the previous three tests in that the Fibonacci sequences are generated concurrently, whereas the sequences for tests 1, 2 and 3 are generated consecutively.
Figure 8.17 An application of a parent node with multiple children. The same DAL specification is used for each child.

The distributed application for test 4 was designed so that each Fibonacci sequence is generated concurrently. This design was achieved by re-using the DAL specification (Fibonacci) three times, one for each of the following sequences:

1 1 2 3 5
1 1 2 3 5 8 13
1 1 2 3 5 8 13 21 34 55
Within this application, one process sends the three lengths (5, 7 and 10) of sequences to the other processes, one length to each process. These processes generate the sequence, and send it back to the first process which displays the sequence.

The root node of the diagram indicates that process `display_sequence` sends the lengths of Fibonacci sequences to three processes: `seq_1`, `seq_2` and `seq_3`. These three processes compute the Fibonacci sequences concurrently, and send\(^1\) the sequences back to `display_sequence` which displays each sequence.

The functionality of the three processes: `seq_1`, `seq_2` and `seq_3`; is specified by the same DAL specification, the one called `Fibonacci`. Therefore, only two DAL specifications are required to specify this application: the DAL specification called `Fibonacci_sequence` for the root node, and the DAL specification called `Fibonacci` is re-used for each of the three leaf nodes.

**Test 4 Results.** The expected number of processes required to instantiate this application is seven: the process called `display_sequence` of the root node is created, and three additional pairs of processes are created for each of the three leaf nodes. Each pair contains process `create_sequence` and process `compute_Fibonacci`. The expected output is the same three Fibonacci sequences as in tests 1, 2 and 3.

The actual number of processes created is seven. This is confirmed by invoking the DASH `status` command. This application displayed the same sequences as expected (Appendix I, Section I.4.3). In addition, the application output shows that the application consisted of the following seven processes, and that it terminated successfully:

- `compute_Fibonacci running on luthin, UNIX PID value equals 7409`,
- `create_sequence running on smaug, UNIX PID value equals 24574`,
- `compute_Fibonacci running on luthin, UNIX PID value equals 7408`,
- `create_sequence running on smaug, UNIX PID value equals 24591`,
- `compute_Fibonacci running on luthin, UNIX PID value equals 7427`,
- `create_sequence running on smaug, UNIX PID value equals 24614`, and
- `display_sequence running on eldacar, UNIX PID value equals 3015`.

The expected and actual results match each other, confirming a successful test.

\(^1\) All three Fibonacci sequences are sent by different processes to the same buffer of process `display_sequence`, confirming that arrays of integers can be sent by several processes to the same destination process.
8.4 Managing Multiple Applications

The DAS system is capable of creating, concurrently running, and terminating many distributed applications. Testing this capability required applications that run for a significant amount of time to allow the operator to produce reports of the system. Three different applications were specified using DAL and DAPI that continually generate a sequence of numbers until terminated by the operator.

1. The first application consists of two processes: P and Q. Process P passes a number to process Q which passes the number plus one back to process P. Upon receiving a number, Process P displays the number, waits for 15 seconds, and passes the number to process Q.

2. The second application is similar to the first, but includes three processes (P, Q and R) running on three computers. Process P passes a number to process Q which passes the number plus one to process R. Upon receiving a number, Process R passes the number to process P which displays the number, waits for 15 seconds, and passes the number to process Q.

3. The third application, again, similar to the first, but includes four processes (P, Q, R and S) running on four computers. Process P passes a number to process Q which passes the number plus one to process R. Upon receiving a number, Process R passes the number to process S. The incoming number at process S is passed to process P which displays the number, waits for 15 seconds, and passes the number to process Q.

These three specifications were then used to create five running applications: one instance of the first application, two instances of the second, and two instances of the third.

The order in which these five applications were created during this test is as follows. For each of the three specifications, an application instance was created and they ran concurrently. The first application instance was terminated, and then the operator created additional instances using the second and third specification. At this time two instances of the second application, and two of the third, were concurrently running on the network.

Managing several distributed applications on a network using the DAS system was successful. A transcript of the DASH II user interface shows that these five application were created, produced expected output, and were terminated successfully. This transcript and DAL and DAPI code for these three applications are presented in Appendix G.
8.5 Conclusion

The tests presented in this chapter confirmed that distributed applications can be specified using the DAL and DAPL languages, and that these applications will execute and produce results as expected. The DAS system was tested with respect to managing several distributed applications concurrently and shown that it was capable of such management.

Additional tests consisted of two groups, the first group of nine tests was used for testing fundamental ways in which processes can be connected, the second group of four tests focused on the hierarchical characteristic of a DAL specification. In particular, the four fundamental ways in which such hierarchies can be formed were tested. In each test the expected results matched results from the running application, demonstrating that distributed applications running on several networked computers can be specified and generated in this way.

These tests demonstrated that the aims of this research were achieved.
9 Conclusion

The aim of this research was to develop a technology that allows a particular class of distributed application to be easily developed. It has been demonstrated that a distributed application can be specified using a combination of two new programming languages, and that such distributed applications can execute on a loosely coupled network. In addition, it is also shown that the creation and execution of such applications can be managed by a runtime system, and that many applications differing in connectivity can be created and managed, not just one.

The proposed technology is formed by two new languages, DAL and DAPL, that have been developed to specify distributed applications. Furthermore, a specification of an application is used by a new compilation system to generate object modules for creating an application instance. This compilation system is a component of another new system, called DAS, that has been produced for creating and concurrently managing such applications.

Currently, designers of computer programs spend a considerable amount of time specifying data and functionality, but little effort is set aside to specify program structure. However, designers of distributed applications cannot ignore the structure of an application. The DAL and DAPL languages allow designers to specify structure independently from data and functionality of a distributed application.

The implementation of the DAS system, the DASH user interface, the new DAL and DAPL programming languages, DAL and DAPL parsers, and the compilation system for the creation and management of a class of distributed application has been completed.

Testing presented in this thesis confirms that distributed applications specified using the DAL and DAPL languages can be created by the DAS system. Testing of fundamental arrangements in which application processes can be connected demonstrated that any structure can be built. In addition, creating applications using hierarchical DAL specifications was achieved. This clearly demonstrates that the proposed technology is feasible.

In summary, the tasks stated in Section 1.3 of the introduction have been solved. The technology has been developed. The aim of this research has been achieved.

9.1 Original Results

Several new software components were developed to achieve the aim of this research such as the DAL and DAPL parsers, compilation system, DAS runtime system, and DASH shell. Although parsers, runtime systems and shells are not new to the computing discipline, the new technology presented in this thesis to develop
distributed applications ensures that the structure of an application can be designed and specified independently from the functionality of the application, and that such distributed applications can be created and executed on a network of computers. The original and significant results of this research in developing this technology are:

**DAL Language.** A new programming language called DAL (Distributed Application Language) has been developed as part of this research. It is used to explicitly specify connectivity of distributed applications with respect to application processes and communication channels, and independently from the application's functionality. The DAL language is also used to specify a set of computers on which these processes execute, the order in which these processes commence execution, and the types of communications channels and buffers. Furthermore, existing DAL specifications can be re-used when developing a new DAL specification to form a hierarchical DAL specification, so enabling modular development of distributed applications.

Several distributed applications were created and executed on a network of computers as part of testing, which clearly shows that the DAL language can be successfully used to specify connectivity of distributed applications.

**DAPL Language.** In addition to the DAL language, another new programming language called DAPL (Distributed Application Process Language) has been developed as part of this research. The DAPL language has been designed to specify four characteristics of a process of the distributed application: functionality, data structures, an interface to the application structure, and high level statements to support network communications.

The DAPL language is used to write DAPL programs where each program specifies the functionality and data structures of a process. These programs can written independently from connectivity, whereas the DAL language is used for writing connectivity specifications.

A DAPL programmer also writes statements defining the interface of each process so that a DAPL program can be inserted into the application's structure while building the application. This interface, and the ability to separate connectivity specifications from functionality specifications, allows DAPL programs to be re-used several times not just within one application but also within other applications.

The DAPL language provides high level statements to declare communications channels and buffers. The runtime system automatically creates and manages these channels and buffers. The DAPL programmer writes high level code for message passing by using a programmer supplied name of a channel, the message, and a library function. Messages are automatically transferred and placed in a specified buffer within the destination process. The DAPL programmer uses only a buffer name and a library function to remove the message from the buffer.
Several additional distributed applications created and executed on a network of computers as part of testing demonstrated that both DAL and DAPL languages facilitate successful development of DAL specification and DAPL programs.

**DAS System.** The DAL and DAPL languages, DAL specifications and DAPL programs cannot on their own instantiate or manage distributed applications. An application management system called DAS (Distributed Application System) was designed and developed as part of this research, to facilitate compilation, instantiation and runtime management of many distributed applications of differing connectivity and functionality. Embedded within the DAS system are three components:

- the DASH user interface providing commands to the operator such as those for compiling, application creation and application termination;
- the compilation system, including DAL and DAPL parsers and translators to C source; and
- a set of internal data structures and operations for storing and managing information about applications running on a network.

The outputs produced by all applications used for testing demonstrate that the DAS system is capable of building applications by linking DAPL programs to a DAL specification, compiling object modules, creating and executing application processes that communicate with each other on a network of computers, and managing several similar and different distributed applications.

## 9.2 Future Work

**Concurrent Creation of a Distributed Application.** The procedure used in this research to create a distributed application involved creating all processes one after the other, followed by creating all communication channels one after the other. Although this procedure is sequential, it was successful in creating a distributed application. An algorithm to create processes and communication channels concurrently (instead of sequentially) can be developed, but not essential for this research, and is set aside as future work.

**Process Allocation Algorithm.** The procedure used in this research to create a process of a distributed application must determine on which computer to create the process. This procedure selects one computer name from a list of names obtained from the DAL specification. The selected computer is the first one listed that is alive. This process allocation algorithm is quite simple, but a more complicated one is not essential for this thesis. A process allocation algorithm dependent on processor loads and other parameters can be integrated into the DAS system as part of future work.
Dynamic Creation of DAS Applications. DAS applications are currently created statically by the operator. Dynamic creation of a DAS application means that a running application can create another application. This is analogous to a UNIX process using the fork() system call to create another process dynamically. It is perceived that dynamic creation of an application can be achieved.

DAPL Preprocessor. The current DAPL grammar requires the programmer to write structure definitions within both sending and receiving programs when a message contain structures. Although the DAPL grammar provides essentials for message passing and therefore structured data can be sent from one process to another, a future version of the DAPL language can contain grammar to support include or inheritance mechanisms to end such duplication, and a DAPL preprocessor can be developed. As an alternative, the compiler could search DAL and DAPL programs for such information.

9.3 Summary

The aim of this research and tasks to accomplish this aim are presented in Chapter 1. The new DAL language solves two of these tasks: specifying connectivity independently from functionality and in a modular fashion to facilitate re-use. The new DAPL language is a solution to two other tasks: specifying functionality independently from connectivity and also as modular components of functionality to build applications. The new DAS system has been developed to solve the tasks of building a complete specification by combining a specified group of DAPL programs with a DAL specification, using this specification to compile object modules, creating an application so that its processes are distributed on a network and can communicate with each other, and managing many distributed applications.

The new DAL and DAPL languages were fundamental in successfully completing this project and achieving the aim of this research. Both languages and the DAS system have been developed as part of this research and shown, within this thesis, to be a significant basis of a new technology for developing distributed applications.
References


References


References


References


References


Appendix A  DAL Grammar

DAL_SPECIFICATION ::= 
  'application'
  APPLICATION_NAME
  APPLICATION_MACHINE_CLAUSE
  APPLICATION_BUFFER_CLAUSE
  APPLICATION_OUTPUT_CLAUSE
  APPLICATION_CREATION_CLAUSE
  PROCESSES_CLAUSE.
  'end' .

APPLICATION_MACHINE_CLAUSE ::= 
  [ 'machines' [MACHINE_NAME {',', MACHINE_NAME}] ] .

APPLICATION_BUFFER_CLAUSE ::= 
  [ 'buffers' [ APPLICATION_BUFFERS ]] .

APPLICATION_BUFFERS ::= 
  TYPE APPLICATION_BUFFER
  [ ',' APPLICATION_BUFFER ]
  [ ';' TYPE APPLICATION_BUFFER { ',' APPLICATION_BUFFER } ] .

APPLICATION_BUFFER ::= 
  BUFFER_NAME ' ; ' PROCESS_NAME .

APPLICATION_OUTPUT_CLAUSE ::= 

APPLICATION_OUTPUTS ::= 
  TYPE APPLICATION_OUTPUT { ' , ' APPLICATION_OUTPUT }
  { ';' TYPE APPLICATION_OUTPUT { ' , ' APPLICATION_OUTPUT } } .

APPLICATION_OUTPUT ::= 
  OUTPUT_NAME ' : ' PROCESS_NAME .

APPLICATION_CREATION_CLAUSE ::= 
  [ 'creation' [PROCESS_NAME{',' PROCESS_NAME}] ].

197
Appendix A. DAL Grammar

```
PROCESSES_CLAUSE ::= [
  'processes' { PROCESS }
].

PROCESS ::= 'process'
           PROCESS_NAME
           REQUIRES_CLAUSE
           PROCESS_MACHINE_CLAUSE
           PROCESS_BUFFER_CLAUSE
           PROCESS_OUTPUT_CLAUSE
           'end'.

REQUIRES_CLAUSE ::= [
  'requires' [ APPLICATION_NAME ]
].

PROCESS_MACHINE_CLAUSE ::= [
  'machine(s)' [ MACHINE_NAME { ',', MACHINE_NAME }]
].

PROCESS_BUFFER_CLAUSE ::= [
  'buffers' [ PROCESS_BUFFERS ]
].

PROCESS_BUFFERS ::=[
  TYPE BUFFER_NAME { ',', BUFFER_NAME }[
  's' TYPE BUFFER_NAME { ',', BUFFER_NAME }]
].

PROCESS_OUTPUT_CLAUSE ::= [
  'outputs' [ PROCESS_OUTPUTS ]
].

PROCESS_OUTPUTS ::=[
  TYPE PROCESS_OUTPUT { ',', PROCESS_OUTPUT }
  ( 's' TYPE PROCESS_OUTPUT { ',', PROCESS_OUTPUT } )
].

PROCESS_OUTPUT ::=[
  OUTPUT_NAME ':' '|
  OUTPUT_NAME ':' BUFFER_NAME ':' PROCESS_NAME
].
```
Appendix B  DAPL Grammar

DAPL_SPECIFICATION ::=  
  { DAPL_PROGRAM } .

DAPL_PROGRAM ::=  
  'program' PROGRAM_NAME  
  STRUCTURES_CLAUSE  
  BUFFERS_CLAUSE  
  OUTPUTS_CLAUSE  
  PRE_MOVE_CLAUSE  
  POST_MOVE_CLAUSE  
  PROGRAM_CODE_CLAUSE  
  'end' .

STRUCTURES_CLAUSE ::=  
  [ 'structures' { DAPL_STRUCTSpecifier ';' } ] .

DAPL_STRUCTSpecifier ::=  
  'struct' C_IDENTIFIER  
  '{'  
  DAPL_STRUCT_DECLARATION { DAPL_STRUCT_DECLARATION }  
  '}' .

DAPL_STRUCT_DECLARATION ::=  
  { DAPL_TYPESpecifier | DAPL_STRUCTSpecifier }  
  DAPL_STRUCT_DECLATOR { ',' DAPL_STRUCT_DECLATOR }  
  ';' .

DAPL_STRUCT_DECLATOR ::=  
  C_IDENTIFIER { '[' INTEGER_CONSTANT ']' } .

BUFFERS_CLAUSE ::=  
  { 'buffers' { BUFFER } } .

BUFFER ::=  
  'buffer'  
  DAPL_TYPESpecifier  
  BUFFER_NAME  
  DATA_ARRIVAL_CODE_CLAUSE  
  PRE_REMOVE_CODE_CLAUSE  
  POST_REMOVE_CODE_CLAUSE  
  'end' .
DATA_ARRIVAL_CODE_CLAUSE ::= 
  [ 'data_arrival_code' DAPL_COMPOUND_STATEMENT ] .

PRE_REMOVE_CODE_CLAUSE ::= 
  [ 'pre_remove_code' DAPL_COMPOUND_STATEMENT ] .

POST_REMOVE_CODE_CLAUSE ::= 
  [ 'post_remove_code' DAPL_COMPOUND_STATEMENT ] .

OUTPUTS_CLAUSE ::= 
  [ 'outputs' { OUTPUT } ] .

OUTPUT ::= 
  'output' 
  DAPL_TYPE_SPECIFIER 
  OUTPUT_NAME 
  PRE_SEND_CLAUSE 
  POST_SEND_CLAUSE 
  'end' .

PRE_SEND_CLAUSE ::= 
  [ 'pre_send_code' DAPL_COMPOUND_STATEMENT ] .

POST_SEND_CLAUSE ::= 
  [ 'post_send_code' DAPL_COMPOUND_STATEMENT ] .

PRE_MOVE_CLAUSE ::= 
  [ 'pre_move_code' DAPL_COMPOUND_STATEMENT ] .

POST_MOVE_CLAUSE ::= 
  [ 'post_move_code' DAPL_COMPOUND_STATEMENT ] .

PROGRAM_CODE_CLAUSE ::= 
  [ 'code' { C_FUNCTION | C_DECLARATION } ] .

DAPL_TYPE_SPECIFIER ::= 
  'struct' C_IDENTIFIER | 
  'char' | 
  'short' | 
  'int' | 
  'long' | 
  'signed' | 
  'unsigned' | 
  'float' | 
  'double' | 
  'void' .
DAPL_COMPOUND_STATEMENT ::=  
  { C_DECLARATION }  
  { C_STATEMENT } .
Appendix C Structures Used by the DAL Parser

Data about a hierarchical DAL specification is collected by the DAL parser during compilation and stored into a tree of data structures. These structures are described below.

C.1 struct tree_node

`struct tree_node` is used to build the a hierarchy of the same structure as the hierarchical DAL specification. Each node is used to store information about the DAL specification, a unique id, and the name of the file in which contains the DAL source.

```c
struct tree_node
{
    struct dal_application *application;
    int application_id;
    char filename[BUFF12];
    struct tree_node *sibling;
    struct tree_node *children;
};
```

C.2 struct dal_application

`struct dal_application` is used to store information about the DAL specification, this information is extracted from the DAL source file by the DAL parser. From the following structure, it can be seen that the following information is stored: the application name, a linked list of machine names, a linked list of interface buffers, a linked list of interface outputs, a linked list of process names associated with the creation clause, a linked list of processes.

```c
struct dal_application
{
    char *name;
    struct dal_machine *machines;
    struct dal_app_buffer *buffers;
    struct dal_app_output *outputs;
    struct dal_creation_process *creation_processes;
    struct dal_process *processes;
};
```
C.3  struct dal_machine

struct dal_machine is used to create a linked list of machine names.

    struct dal_machine
    {
        char *name;
        struct dal_machine *next;
    };

C.4  struct dal_app_buffer

struct dal_app_buffer is used to create a linked list of information about each interface buffer. Each node of the list contains the buffer’s name, the type, and the name of the process in which the buffer is declared.

    struct dal_app_buffer
    {
        char *name;
        char *type;
        char *host_process_name;
        struct dal_app_buffer *next;
    };

C.5  struct dal_app_output

struct dal_app_output is used to create a linked list of information about interface outputs. Each node of the list contains the output’s name, the type, and the name of the process in which the output is declared.

    struct dal_app_output
    {
        char *name;
        char *type;
        char *source_process_name;
        struct dal_app_output *next;
    };
C.6 \textbf{struct dal_creation_process}

\textit{struct dal_creation_process} is used to create a linked list of process names. These names are the ones specified within the creation clause of the DAL specification.

\begin{verbatim}
struct dal_creation_process
{
    char *name;
    struct dal_creation_process *next;
};
\end{verbatim}

C.7 \textbf{struct dal_process}

\textit{struct dal_process} is used to create a linked list of process information. Each node contains the following information about each process: the process name, the name of a required application, i.e., the child DAL, specification of this process, a linked list of machines names, a linked list of buffer information, a linked list of output information.

\begin{verbatim}
struct dal_process
{
    char *name;
    char *req_application;
    struct dal_machine *machines;
    struct dal_buffer *buffers;
    struct dal_output *outputs;
    struct dal_process *next;
};
\end{verbatim}

C.8 \textbf{struct dal_buffer}

\textit{struct dal_buffer} is used to create a linked list of information about each buffer of a process. Each node contains the buffer’s name and type.

\begin{verbatim}
struct dal_buffer
{
    char *name;
    char *type;
    struct dal_buffer *next;
};
\end{verbatim}
C.9  \textbf{struct dal_output}

\textit{struct dal_output} is used to create a linked list of output information. Each node contains the output's name, type, the names of the destination buffer and destination process.

\begin{verbatim}
struct dal_output
{
    char *name;
    char *type;
    char *dest_buffer_name;
    char *dest_process_name;
    struct dal_output *next;
};
\end{verbatim}
Appendix D  Structures Used by the DAPL Parser

The DAPL parser extracts information from a DAPL program and stores this information into the following structures.

D.1 struct dapl_program_node

The `struct dapl_program_node` is used to build a linked list of `struct dapl_program` pointers, each pointer refers to data of a DAPL program that is extracted by the DAPL parser.

```c
struct dapl_program_node
{
    struct dapl_program *program;
    struct dapl_program_node *next;
};
```

D.2 struct dapl_program

The `struct dapl_program` is used to store information about a DAPL program, this information is extracted from the DAPL source file by the DAPL parser. The following information is stored: the program name, the name of the file contain this DAPL program, a linked list of information that describe structures, the location of these structures within the DAPL file, a linked list of buffers, a linked list of outputs, the location of C code fragments within the DAPL file.

```c
struct dapl_program
{
    char *name;
    char *source_filename;
    struct dapl_structure *structures;
    struct dapl_code structures_code;
    struct dapl_buffer *buffers;
    struct dapl_output *outputs;
    struct dapl_code pre_move_code;
    struct dapl_code post_move_code;
    struct dapl_code program_code;
};
```
D.3 struct dapl_structure

*struct dapl_structure* is used to store information that describes a structure. It is also allows the creation of a linked list to store many structure descriptions. The struct tag is stored in the name field. The type of each data member of the structure is stored in a linked list, the order of the data members matches the order of the linked list elements.

```c
struct dapl_structure
{
    char *name;
    struct dapl_data *info;
    struct dapl_structure *next;
};
```

D.4 struct dapl_buffer

*struct dapl_buffer* is used to store the following information about a buffer: the buffer name and type, and the locations of three C code fragments within the DAPL source file.

```c
struct dapl_buffer
{
    char *name;
    char *type;
    struct dapl_code data_arrival_code;
    struct dapl_code pre_remove_code;
    struct dapl_code post_remove_code;
    struct dapl_buffer *next;
};
```

D.5 struct dapl_output

*struct dapl_output* is used to store the following information about an output: the output name and type, and the locations of two C code fragments within the DAPL source file.

```c
struct dapl_output
{
    char *name;
    char *type;
    struct dapl_code pre_send_code;
    struct dapl_code post_send_code;
    struct dapl_output *next;
};
```
D.6 **struct dapl_code**

*struct dapl_code* is used to store the location and size of a code fragment within a DAPI file. *offset* is used to store the offset of this code fragment from the start of the file, in bytes. *length* is used to store the size of this code fragment, in bytes.

```c
struct dapl_code
{
    int offset;
    int length;
};
```

D.7 **struct type_data**

*struct type_data* is used to store information about a type. *type_id* is used to store an integer representing a fundamental type such as char, int or float. If the type is a structure, -1 is used. The header file type_id.h contains the following type_id values.

```c
#define UNDEFINED_TYPE -1
#define VOID_TYPE 0
#define CHAR_TYPE 1
#define SHORT_TYPE 2
#define INT_TYPE 3
#define LONG_TYPE 4
#define FLOAT_TYPE 5
#define DOUBLE_TYPE 6
#define SIGNED_TYPE 7
#define UNSIGNED_TYPE 8
```

The *struct_id* field stores an identifier of the structure, -1 is store for a fundamental type. The type size in bytes is stored. If the type is an array, the number of array elements is stored. This structure can be used to form a linked list of, say, N elements to support the description of a structure of N data members.

```c
struct type_data
{
    int type_id;
    int struct_id;
    int type_size;
    int num_elements;
    struct type_data *next;
};
```
Appendix E  Identification Numbers of Fundamental Types

Identification numbers associated with fundamental types such as char and int are defined in the DAS header file called type_id.h in which the contents is:

```c
#define UNDEFINED_TYPE -1
#define VOID_TYPE 0
#define CHAR_TYPE 1
#define SHORT_TYPE 2
#define INT_TYPE 3
#define LONG_TYPE 4
#define FLOAT_TYPE 5
#define DOUBLE_TYPE 6
#define SIGNED_TYPE 7
#define UNSIGNED_TYPE 8
```
Appendix F  Listing of the Header File das_structs.h

#include <unistd.h>

/* structures for the das command */
#define CLOSE_COMMAND 0
#define CREATEAPP_COMMAND 1
#define CREATEPROC_COMMAND 2
#define CREATEPROC_V2_COMMAND 3
#define EXITAPP_COMMAND 4
#define EXITPROC_COMMAND 5
#define MOVE_COMMAND 6
#define OPEN_COMMAND 7
#define STATUS_COMMAND 8

#define ACTIVE 0
#define PENDING 1

struct close_command
{
    char *flow_name;
    char *source_process_name;
};

struct createapp_command
{
};

struct createproc_command
{
    char *process_name;
    char *host_name;
};

struct exitapp_command
{
};

struct exitproc_command
{
    char *process_name;
};
struct move_command
{
    char *process_name;
    char *host_name;
};

struct open_command
{
    char *flow_name;
    char *source_process_name;
};

struct status_command
{
};

struct das_command
{
    int command;
    int state;
    union
    {
        struct close_command close;
        struct createapp_command createapp;
        struct createproc_command createproc;
        struct createproc_command createproc_v2;
        struct exitapp_command exitapp;
        struct exitproc_command exitproc;
        struct move_command move;
        struct open_command open;
        struct status_command status;
    } args;
    unsigned long id;
    struct das_command *next;
};

struct das_commands
{
    struct das_command *head;
    struct das_command *tail;
};

/* end structures for the das command */
struct das_host
{
    char *name;
    struct das_host *next;
};

struct das_DAPL_program
{
    struct dapl_program *dapl_program;
    struct das_DAPL_program *next;
};

struct das_DAL_application
{
    char *source_directory;
    struct dal_application *dal_application;
    struct das_DAPL_program *das DAPL programs;
    struct das_DAL_application *next;
};

struct das_output
{
    char *name;
    char *dest_process_name;
    int port;
    char source_open, destination_open;
    struct das_output *next;
};

struct das_output_pre_move
{
    char *name;
    char *source_process_name;
    char *dest_process_name;
    struct das_output_pre_move *next;
};

struct das_process_pre_move
{
    int state;
    struct das_output_pre_move *open_flows;
};
struct das_process
{
    char *name;
    struct das_host *machines;
    struct das_output *outputs;
    char *host;
    int port;
    pid_t pid;
    int state;
    char *destination_host;
    int exit_status;
    struct das_process *pre_move *pre_move_data;
    struct das_process *next;
};

struct das_application
{
    char *name;
    unsigned long id;
    int state;
    struct das_host *machines;
    struct das_creation_process *creation_list;
    struct das_process *processes;
    struct das_application *next;
    struct das_commands das_commands;
};

struct das
{
    struct das_host *das_hosts;
    struct das.DAL.application *das.DAL.applications;
    struct das_application *das_applications;
    char *das_hostname;
    int das_portnum;
    int das_sd;
    int das_state;
    unsigned long last_app_id;
    unsigned long last_command_id;
};

/* add prototypes */
extern struct das_application *add_das_application(
    struct das_application *,
    struct das_application *);

extern struct das.DAL.application *add_das.DAL.application(
    struct das.DAL.application *,
    struct dal.application *);
extern struct das_DAPL_program *add_das_DAPL_program(
    struct das_DAPL_program *,
    struct dapi_program *);
extern struct das_host *add_das_host(struct das_host *, char *);
extern struct das_output *add_das_output(struct das_output *,
    char *, char *);
extern struct das_process *add_das_process(struct das_process *,
    char *);
extern struct das_output_pre_move *add_pre_move_flow_data(
    struct das_output_pre_move *,
    struct das_output *, char *);

/* add das commands prototypes */
extern struct das_command *add_close_das_command(
    struct das_commands *,
    char *, char *,
    unsigned long );
extern struct das_command *add_createapp_das_command(
    struct das_commands *,
    unsigned long );
extern struct das_command *add_createproc_das_command(
    struct das_commands *,
    char *, char *,
    unsigned long );
extern struct das_command *add_createproc_v2_das_command(
    struct das_commands *,
    char *, char *,
    unsigned long );
extern struct das_command *add_exitapp_das_command(
    struct das_commands *,
    unsigned long );
extern struct das_command *add_exitproc_das_command(
    struct das_commands *,
    char *, unsigned long );
extern struct das_command *add_move_das_command(
    struct das_commands *,
    char *, char *,
    unsigned long );
extern struct das_command *add_open_das_command(
    struct das_commands *,
    char *, char *,
    unsigned long );
extern struct das_command *add_status_das_command(
    struct das_commands *,
    unsigned long );
extern struct das *das_create();
extern struct das_application *create_das_application{
  struct das *};
extern unsigned long create_das_id(struct das_application *,
  unsigned long *);
extern struct das_process_pre_move *create_pre_move_data{
  struct das_process *
  struct das_application *};

extern struct das_application *delete_das_application{
  struct das_application *,
  struct das_application *};
extern int delete_active_das_command(struct das_commands *);
extern int delete_das_command(struct das_commands *,
  unsigned long);
extern struct das_DAL_application *delete_das_DAL_application{
  struct das_DAL_application *,
  struct das_DAL_application *};
extern struct das_host *delete_das_host(struct das_host *,
  struct das_host *);
extern struct das_DAPI_program *delete_das_DAPI_program{
  struct das_DAPI_program *,
  struct das_DAPI_program *};

extern void evaluate_pending_command(struct das *,
  struct das_application *);

extern void free_das(struct das *);
extern void free_das_application(struct das_application *);
extern void free_das_command(struct das_command *);
extern void free_das_DAL_application{
  struct das_DAL_application *
};
extern void free_das_host(struct das_host *);
extern void free_das_DAPI_program(struct das_DAPI_program *);
extern void free_das_output_pre_move{
  struct das_output_pre_move *
};
extern void free_das_process(struct das_process *);
extern void free_pre_move_data(struct das_process_pre_move *);}
Appendix F. Listing of the Header File das structs.h

/* get prototypes */
extern struct das_application *get_das_application(
    struct das_application *,
    unsigned long);
extern struct das DAL_application *get_das.DAL_application(
    struct das.DAL_application *,
    char *);
extern struct das_host *get_das_host(struct das_host *, char *);
extern struct das_DAPL_program *get_das_DAPL_program(
    struct das_DAPL_program *,
    char *);
extern struct das_output *get_das_output(struct das_output *,
    char *);
extern struct das_process *get_das_process(struct das_process *,
    char *);
extern unsigned long get_next_command_id(struct das *);
extern struct das_host *get_next_das_host(struct das *,
    struct das_application *,
    struct das_process *);
extern struct das_command *get_active_command(
    struct das_application *);
extern struct das_command *get_das_command(
    struct das_application *,
    unsigned long);
extern struct das_command *get_pending_command(
    struct das_application *);

/* report prototypes */
extern void das.DAL_app_report(struct das.DAL_application *);
extern void das.hosts_report(struct das.host *);
extern void das.DAPLprog_report(struct das.DAPL_program *);
extern void das_report(struct das *);
Appendix G  Testing Applications

The transcript of the test described in Section 8.4 is listed in Section G.1. This test requires DAL specification and DAPI programs for three applications. These are listed in Section G.2 to Section G.7.

G.1 Transcript

dash >>addhost aldcar luthin elwing bilbo
dash >>addapp Test_2p.dal
dash >>addprog Test_2p.dapl Test_2p
dash >>compile Test_2p
dash >>addapp Test_3p.dal
dash >>addprog Test_3p.dapl Test_3p
dash >>compile Test_3p
dash >>addapp Test_4p.dal
dash >>addprog Test_4p.dapl Test_4p
dash >>compile Test_4p
dash >>createapp Test_2p
  Test_2p: AppID - 1; i = 201
  Test_2p: AppID - 1; i = 202

dash >>createapp Test_3p
dash >>createapp Test_4p
  Test 3p: AppID - 2; i = 3001
  Test 2p: AppID - 1; i = 203
  Test 4p: AppID - 3; i = 40001
  Test 3p: AppID - 2; i = 3002
  Test 2p: AppID - 1; i = 204
  Test 4p: AppID - 3; i = 40002
  Test 3p: AppID = 2; i = 3003
  Test 2p: AppID - 1; i = 205
  Test 4p: AppID - 3; i = 40003
  Test 3p: AppID = 2; i = 3004
  Test 2p: AppID - 1; i = 206
  Test 4p: AppID = 3; i = 40004
  Test 3p: AppID - 2; i = 3005
  Test 2p: AppID - 1; i = 207
  Test 4p: AppID - 3; i = 40005
  Test 3p: AppID = 2; i = 3006
  Test 2p: AppID - 1; i = 208
  Test 4p: AppID = 3; i = 40006
  Test 3p: AppID = 2; i = 3007
Test_2p: AppID = 1: i = 209
Test_4p: AppID = 3: i = 40007
Test_3p: AppID = 2: i = 3000
Test_2p: AppID = 1: i = 210
Test_4p: AppID = 3: i = 40008
Test_3p: AppID = 2: i = 3009
Test_2p: AppID = 1: i = 211
Test_4p: AppID = 3: i = 40009
Test_3p: AppID = 2: i = 3010
Test_2p: AppID = 1: i = 212

dash >> status 1
Test_2p <id=1, RUNNING>
p0_0 <host=eldacar, pid=10811, port=6002, exit_status=0,
    RUNNING> eldacar
    o0_0_0 port=6003 (open, open)
p1_0 <host=luthin, pid=12323, port=6002, exit_status=0,
    RUNNING> luthin
    o0 1 0 port=34953 (open, open)
dash >> status 2
Test_3p <id=2, RUNNING>
p0_0 <host=eldacar, pid=10841, port=6003, exit_status=0,
    RUNNING> eldacar
    o0 0 0 port=6005 (open, open)
p1_0 <host=luthin, pid=12334, port=6004, exit_status=0,
    RUNNING> luthin
    o0 1 0 port=6003 (open, open)
p2_0 <host=elwing, pid=398, port=6002, exit_status=0,
    RUNNING> elwing
    o0 2 0 port=608 (open, open)
dash >> status 3
Test_4p <id=3, RUNNING>
p0_0 <host=eldacar, pid=10873, port=6005, exit_status=0,
    RUNNING> eldacar
    o0 0 0 port=6007 (open, open)
p1_0 <host=luthin, pid=12345, port=6006, exit_status=0,
    RUNNING> luthin
    o0 1 0 port=6005 (open, open)
p2_0 <host=elwing, pid=409, port=6004, exit_status=0,
    RUNNING> elwing
    o0 2 0 port=6003 (open, open)
p3 0 <host=bilbo, pid=18032, port=6002, exit_status=0,
    RUNNING> bilbo
    o0 3 0 port=6006 (open, open)
Test_4p: AppID = 3: i = 40010
Test_3p: AppID = 2: i = 3011
Appendix G. Testing Multiple Applications

Test_2p: AppID = 1; i = 213
Test_4p: AppID = 3; i = 40011
Test_3p: AppID = 2; i = 3012
Test_2p: AppID = 1; i = 214

dash >>exitapp 1
Test_3p: AppID = 3; i = 40012
Test_2p: AppID = 2; i = 3013
Test_2p: AppID = 1; i = 215
Test_4p: AppID = 3; i = 40013
Test_3p: AppID = 2; i = 3014
Test_2p: AppID = 1; i = 216

proc 'p0_0' of app 'Test_2p, 1' terminated
proc 'p1 0' of app 'Test_2p, 1' terminated
app 'Test_2p, 1' TERMINATED
Test_4p: AppID = 3; i = 40014
Test_3p: AppID = 2; i = 3015

dash >>status

das <host-_oldacar, port-_6001, RUNNING>

    oldacar luthin oldwing bilbo

    ID  APP NAME  STATE
    2   Test_3p  RUNNING
    3   Test_4p  RUNNING

dash >>createapp Test_3p
Test_4p: AppID = 3; i = 40015
Test_3p: AppID = 2; i = 3016
Test_4p: AppID = 3; i = 40016
Test_3p: AppID = 2; i = 3017
Test_3p: AppID = 4; i = 3001

dash >>createapp Test_4p
Test_3p: AppID = 2; i = 3018
Test_4p: AppID = 3; i = 40017
Test_3p: AppID = 4; i = 3002
Test_4p: AppID = 5; i = 40001
Test_3p: AppID = 2; i = 3019
Test_4p: AppID = 3; i = 40018
Test_3p: AppID = 4; i = 3003
Test_4p: AppID = 5; i = 40002
Test_3p: AppID = 2; i = 3020
Test_4p: AppID = 3; i = 40019
Appendix G. Testing Multiple Applications

dash $>>$status
das <host=eldacar, port=6001, RUNNING>
eldacar luthin elwing bilbo

<table>
<thead>
<tr>
<th>ID</th>
<th>APP_NAME STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>Test_3p RUNNING</td>
</tr>
<tr>
<td>3</td>
<td>Test_4p RUNNING</td>
</tr>
<tr>
<td>4</td>
<td>Test_3p RUNNING</td>
</tr>
<tr>
<td>5</td>
<td>Test_4p RUNNING</td>
</tr>
</tbody>
</table>

Test_3p: AppID = 4: i = 3004
Test_4p: AppID = 5: i = 40003
Test_3p: AppID = 2: i = 3021
Test_4p: AppID = 3: i = 40020
Test_3p: AppID = 4: i = 3005
dash $>>$exitapp 2
Test_4p: AppID = 5: i = 40004
Test_3p: AppID = 2: i = 3022
proc 'p0_0' of app 'Test_3p, 2' terminated
proc 'p1_0' of app 'Test_3p, 2' terminated
proc 'p2_0' of app 'Test_3p, 2' terminated
app 'Test_3p, 2' TERMINATED
Test 4p: AppID = 3: i = 40021
Test 3p: AppID = 4: i = 3006
dash $>>$status
das <host=eldacar, port=6001, RUNNING>
eldacar luthin elwing bilbo

<table>
<thead>
<tr>
<th>ID</th>
<th>APP_NAME STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>Test_4p RUNNING</td>
</tr>
<tr>
<td>4</td>
<td>Test_3p RUNNING</td>
</tr>
<tr>
<td>5</td>
<td>Test_4p RUNNING</td>
</tr>
</tbody>
</table>

dash $>>$exitapp 3
Test_4p: AppID = 5: i = 40005
Test_3p: AppID = 4: i = 3007
Test_4p: AppID = 3: i = 40022
proc 'p0_0' of app 'Test_4p, 3' terminated
proc 'p1_0' of app 'Test_4p, 3' terminated
proc 'p2_0' of app 'Test_4p, 3' terminated
proc 'p3_0' of app 'Test_4p, 3' terminated
app 'Test_4p, 3' TERMINATED
Test_4p: AppID = 5: i = 40006
dash $>>$status
das <host=eldacar, port=6001, RUNNING>
eldacar luthin elwing bilbo

<table>
<thead>
<tr>
<th>ID</th>
<th>APP_NAME STATE</th>
</tr>
</thead>
<tbody>
<tr>
<td>4</td>
<td>Test_3p RUNNING</td>
</tr>
<tr>
<td>5</td>
<td>Test_4p RUNNING</td>
</tr>
</tbody>
</table>

Test_3p: AppID = 4: i = 3008
dasm >>exitapp 4
Test_4p: AppID = 5: i = 40008
Test_3p: AppID = 4: i = 3009
proc 'p0_0' of app 'Test_3p', 4' terminated
proc 'p1_0' of app 'Test_3p', 4' terminated
proc 'p2_0' of app 'Test_3p', 4' terminated
app 'Test_3p', 4' TERMINATED

dasm >>status

das <host=eldacar, port=6001, RUNNING>
edacar luthin eiwing bilbo

<table>
<thead>
<tr>
<th>ID</th>
<th>APP_NAME</th>
<th>STATUS</th>
</tr>
</thead>
<tbody>
<tr>
<td>5</td>
<td>Test_4p</td>
<td>RUNNING</td>
</tr>
</tbody>
</table>

Test_4p: AppID = 5: i = 40008
Test_4p: AppID = 5: i = 40009


dasm >>exitapp 5
Test_4p: AppID = 5: i = 40010
proc 'p0_0' of app 'Test_4p', 5' terminated
proc 'p1_0' of app 'Test_4p', 5' terminated
proc 'p2_0' of app 'Test_4p', 5' terminated
proc 'p3_0' of app 'Test_4p', 5' terminated
app 'Test_4p', 5' TERMINATED

dasm >>status

das <host=eldacar, port=6001, RUNNING>
edacar luthin eiwing bilbo

dasm >>quit

G.2 DAL Specification of Application Test_2p

application Test_2p

processes

process P
machines eldacar
buffers int buf
outputs int x;buf;P
end

process Q
machines luthin
buffers int buf
outputs int y;buf;P
end
end
G.3 DAPL Programs of Application Test_2p

program  P
buffers
  buffer
    int buf

data_arrival_code
  #define get_int(array, index) {((int *)array)[index]}
  struct buffer_data *element;
  int i;

  element = DA_remove(buf);
  i = get_int(element->array, 0);
  printf("Test_2p: AppID = %d: i = %d\n", das_app_id, i);
  DA_free(element);
  sleep(15);
  DA_send(x, &i, 1);
end

outputs
  output
    int x
end

code
  main()
  {
    int i = 200;

    DA_send(x, &i, 1);
  }
end

program  0
buffers
  buffer
    int buf

data_arrival_code
  #define get_int(array, index) {((int *)array)[index]}
  struct buffer_data *element;
  int i;

  element = DA_remove(buf);
  i = get_int(element->array, 0);
  DA_free(element);
G.4 DAL Specification of Application Test_3p

application Test_3p
processes
  process P
  machines eldscar
  buffers int buf
  outputs int x:buf:Q
end

  process Q
  machines luthin
  buffers int buf
  outputs int y:buf:R
end

  process R
  machines elwing
  buffers int buf
  outputs int z:buf:P
end
end

G.5 DAPL Programs of Application Test_3p

program p
buffers
  buffer
    int buf

data_arrival_code
  #define get_int(array, index) {{{(int *)array}[index]}}
  struct buffer_data *element;
  int i;
element = DA_remove(buf);
i = get_int(element->array, 0);
printf("Test_3p: AppID = %ld: i = %d\n", das_app_id, i);
DA_free(element);
sleep(15);
DA_send(x, &i, 1);
}
end
outputs
output
int x
end
code
main()
{
    int i = 3000;

    DA_send(x, &i, 1);
}
end
program
Q
buffers
buffer
int buf
data_arrival_code
#define get_int(array, index) (*((int *)array)[index])
struct buffer_data *element;
int i;

element = DA_remove(buf);
i = get_int(element->array, 0);
DA_free(element);
i++;
DA_send(y, &i, 1);
end
outputs
output
int y
end
end


```
program
  R
buffers
  buffer
    int buf

data_arrival_code
  #define get_int(array, index) (((int *)array)[index])
  struct buffer_data *element;
  int i;

  element = DA_remove(buf);
  i = get_int(olomont->array, 0);
  DA_free(element);
  DA_send(z, &i, 1);
end
outputs
  output
    int z
end
end

G.6 DAL Specification of Application Test_4p

application Test_4p
processes
  process P
    machines eldacar
  buffer buffers int buf
  outputs int a:buf:Q
end

  process Q
    machines luthin
  buffer buffers int buf
  outputs int b:buf:R
end

  process R
    machines elwing
  buffer buffers int buf
  outputs int r:buf:S
end
```
process s
machines bilbo
buffers int buf
outputs int d:buf:
end

G.7 DAPL Programs of Application Test_4p

program p
buffers
buffer
    int buf
data_arrival_code
    #define get_int(array, index) {(*int *)array[index]}
    struct buffer_data *element;
    int i;

    element = DA_remove(buf);
    i = get_int(element->array, 0);
    printf("Test_4p: AppID - %ld: i - %d\n", DA_app_id, i);
    DA_free(element);
    sleep(10);
    DA_send(a, &i, 1);
end
outputs
output
    int a
end
code
main()
{
    int i = 40000;

    DA_send(a, &i, 1);
}
end
program Q
buffers
buffer
  int buf
end

data_arrival_code
#define get_int(array, index) (((int *)array)[index])
struct buffer_data *element;
int i;

element = DA_remove(buf);
i = get_int(element->array, 0);
DA_free(element);

++;
DA_send(b, &i, 1);
end

outputs
output
  int b
end
end

program R
buffers
buffer
  int buf
end

data_arrival_code
#define get_int(array, index) (((int *)array)[index])
struct buffer_data *element;
int i;

element = DA_remove(buf);
i = get_int(element->array, 0);
DA_free(element);
DA_send(c, &i, 1);
end

outputs
output
  int c
end
end
program
  s
buffers
  buffer
    int buf
  data_arrival_code
    #define get_int(array, index) (((int *)array)[index])
    struct buffer_data *element;
    int i;

    element = DA_remove(buf);
    i = get_int(element->array, 0);
    DA_free(element);
    wa_send(d, &i, 1);
end
  outputs
    output
      int d
end
end
Appendix H  DAL  Specifications  and DAPL Programs: Testing Connectivity

Testing connectivity of distributed applications is presented in Section 8.2 Testing Fundamental Connectivity Arrangements. The DAL specifications, DAPL programs and application outputs of such test applications are presented in this appendix.

H.1  Test 1: Single Output from One Process to Single Buffer

This test is described in Section 8.2.1 and requires one DAL specification and two DAPL programs. These specification, programs, and application output are listed in the following subsections.

H.1.1  DAL Specifications

```
application Test 1
processes
  process P
  machines smaug
  output x:buf:Q
end

process Q
  machines eldacar
  buffers int buf
end
```

H.1.2  DAPL Programs

```
program P
  outputs
    output int x
end
```
code
    main()
    {
        int i = 123;

        DA_send(x, &i, 1);
    }
end

program
    Q
buffers
    buffer
        int buf
    data_arrival_code
        #define get int(array, index) (((inL *)array)[index])
        struct buffer_data *element;

        element = DA_recv(buf);
        printf("%d\n", get(element->array, 0));
        DA_free(element);
        DA_exitapp(0);
    end
end

H.1.3 Application Output

dash>>addhost smaug eldabar luthin olwing
dash>>addapp Test_1.dal
dash>>addprog Test_1.dapl Test_1
dash>>compile Test_1
dash>>createapp Test_1
dash>>123
status 1
Test_1 <id=1, RUNNING>
  P <host=smaug, pid=21953, port=6002, exit status=0,
      RUNNING> smaug
      x port=6004 (open, open)
Q <host=eldabar, pid=2650, port=6002, exit status=0,
  RUNNING> eldabar
proc 'P' of app 'Test_1', 1 terminated
proc 'Q' of app 'Test_1', 1 terminated
app 'Test_1', 1 TERMINATED
H.2 Test 2: Two Outputs from One Process to a Single Buffer

This test is described in Section 8.2.2 and requires one DAL specification and two DAPI programs. These specification, programs, and application output are listed in the following subsections.

H.2.1 DAL Specifications

```plaintext
application Test_2

processes

process P
machines smug
outputs int x:buf:Q;
    int y:buf:Q
end

process Q
machines cldacs
buffers int buf
end

end
```

H.2.2 DAPI Programs

```plaintext
program P

outputs

output
    int x
end
output
    int y
end

code

main()
{
    int i = 123, j = 456;
    DA_send(x, &i, 1);
    DA_send(y, &j, 1);
}
end
```
program
0
buffers
  buffer
    int buf
  data_arrival_code
    #define get_int(array, index) {((int *)array)[index]}
    struct buffer_data *element;
    static int tally = 0;
    element = DA_remove(buf);
    printf("%d\n", get_int(element->array, 0));
    DA_free(element);
    if (tally > 2)
      DA_exitapp(0);
end
end

H.2.3 Application Output

dash>>addhost smaug eldcar luthin elwing
dash>>addapp Test_2.dal
dash>>addprog Test_2.dapl Test_2
dash>>compile Test_2
dash>>createapp Test_2
dash>>123
456
status 1
Test_2 <id=1, RUNNING>
  P <host=smaug, pid=22048, port=6002, exit_status=0, 
      RUNNING> smaug
    x port=6003 (open, open)
    y port=6003 (open, open)
  Q <host=eldcar, pid=2668, port 6002, exit_status=0, 
     RUNNING> eldcar
  proc 'P' of app 'Test_2, 1' terminated
  proc 'Q' of app 'Test_2, 1' terminated
  app 'Test_2, 1' TERMINATED
H.3 Test 3: Two Outputs from Two Processes to a Single Buffer

This test is described in Section 8.2.3 and requires one DAL specification and three DAPL programs. These specification, programs, and application output are listed in the following subsections.

H.3.1 DAL Specifications

```plaintext
application Test_3
processes
    process P
    machines smaug
    outputs int x:buf:Q
    end

    process Q
    machines eldagar
    buffers int buf
    end

    process R
    machines luthin
    outputs int y:buf:Q
    end

end
```

H.3.2 DAPL Programs

```plaintext
program P
outputs
    output
        int x
    end

code
    main()
    {
        int i = 123;
        DA_send(x, &i, 1);
    }
end
```
program
Q
buffers
buffer
int buf

data_arrival_code
#define get_int(array, index) (*((int *)array)[index])
struct buffer, data *element;
static int tally = 0;

element = DA_remove(buf);
printf("%d\n", get_int(element->array, 0));
DA_free(element);
if(++tally >= 2)
   DA_exitapp(0);

end
end

program
R
outputs
output
   int y
end
code
main()
{
   int j = 456;
   DA_send(y, &j, 1);
}
end

H.3.3 Application Output

dash>>addhost smaug eldarcar luthin elwing
dash>>addapp Test_3.dal
dash>>addprog Test_3.dapl Test_3
dash>>compile Test_3
dash>>createapp Test_3
dash>>123
456
status 1
Test_3 <id=1, RUNNING>
P <host=smaug, pid=22149, port=6002, exit status=0, RUNNING> smaug
   x port=6003 (open, open)
Appendix H. DAL Specifications and DAPL Programs: Testing Connectivity

Q <host=eldacar, pid=2683, port=6002, exit_status=0, 
    RUNNING> eldacar
R <host=luthin, pid=5936, port=6002, exit_status=0, 
    RUNNING> luthin
    y port=6004 {open, open}
proc 'F' of app 'Test_3, 1' terminated
proc 'Q' of app 'Test_3, 1' terminated
proc 'R' of app 'Test_3, 1' terminated
app 'Test_3, 1' TERMINATED

H.4 Test 4: Two Outputs for Bidirectional Data Transfer

This test is described in Section 8.2.4 and requires one DAL specification and two 
DAPL programs. These specification, programs, and application output are listed in 
the following subsections.

H.4.1 DAL Specifications

application Test_4
processes
    process P
    machines smauq
    buffers int buf
    outputs int x,buf,Q
    end

    process Q
    machines eldacar
    buffers int buf
    outputs int y,buf,P
    end

end

H.4.2 DAPL Programs

program P
buffers
    buffer
        int buf
    data_arrival_code
        #define get_int(array, index) {((int *)array)[index]}
        struct buffer_data *element;
        element -> block = DA_remove(buf);
        printf("%d\n", got_int(element->array, 0));
DA_free(element);
DA_exitapp(0);

end

outputs
output
int x
end

code
main()
|
int i = 123;

DA_send(x, &i, 1);
}
end

program Q
buffers
buffer
int buf
data_arrival_code

#define get_int(array, index) (((int *)array)[index])
struct buffer_data *element;
int i;

element = DA_remove(buf);
i = get_int(element->array, 0);
DA_free(element);

printf("%d\n", i);
DA_send(y, &i, 1);
end

outputs
output
int y
end

H.4.3 Application Output

dash>>addhost smaug eldcar luthin elwing
dash>>addapp Test_4.dal
dash>>addprog Test_4.dapl Test_4
dash>>compile Test_4
dash>>createapp Test_4
H.5 Test 5: Two Outputs from One Process to Two Buffers

This test is described in Section 8.2.5 and requires one DAL specification and two DAPI programs. These specification, programs, and application output are listed in the following subsections.

H.5.1 DAL Specifications

```plaintext
class Test 5

process P
    machines smaug
    output
        int x:buf1:Q;
        int y:buf2:Q
    end

process Q
    machines eldacar
    buffers int buf1, buf2
end
```

H.5.2 DAPI Programs

```plaintext
program P
    outputs
        output int x
    end
    output int y
end
```
main()
{
    int i = 123, j = 456;
    DA_send(x, &i, 1);
    DA_send(y, &j, 1);
}
end

program
Q
buffers
buffer
    int buf1
data_arrival_code
    incoming_data("buf1", DA_remove(buf1));
end

buffer
    int buf2
data_arrival_code
    incoming_data("buf2", DA_remove(buf2));
end
code
#define get_int(array, index) ((int *)array)[index]
int buf1_tally, buf2_tally;

main()
{
    buf1_tally = buf2_tally = 0;
}

incoming_data(char *name, struct buffer_data *element)
{
    if(strcmp(name, "buf1") == 0)
        buf1_tally++;
    else if(strcmp(name, "buf2") == 0)
        buf2_tally++;
    printf("%s, %d\n", name, get_int(element->array, 0));
    DA_free(element);
    if(buf1_tally > 1 && buf2_tally >= 1)
        DA_exitapp(0);
}
end
H.5.3 Application Output

dash>>addhost smaug aldacar luthin elwing
dash>>addapp Test_5.dal
dash>>addprog Test_5.dapl Test_5
dash>>compile Test_5
dash>>createapp Test_5

dash>>buf1, 123
buf2, 456

status 1

Test_5 <id=1, RUNNING>
P <host=smaug, pid=22910, port=6002, exit_status=0,
  RUNNING> smaug
  x port=6003 (open, open)
  y port 6003 (open, open)
Q <host=eldacar, pid=2054, port 6002, exit_status=0,
  RUNNING> eldacar
proc 'P' of app 'Test_5, 1' terminated
proc 'Q' of app 'Test_5, 1' terminated
app 'Test_5, 1' TERMINATED

H.6 Test 6: Two Outputs from Two Processes to Two Buffers

This test is described in Section 8.2.6 and requires one DAL specification and three DAPL programs. These specification, programs, and application output are listed in the following subsections.

H.6.1 DAL Specifications

application Test_6

processes

process P
  machines smaug
outputs int x:buf1:Q
end

process Q
  machines eldacar
  buffers int buf1;
      int buf2
end

process R
  machines luthin
  outputs int y:buf2:Q
end
**H.6.2 DAPL Programs**

```dapl
program P
outputs
output
  int x
end
code
main()
{  int i = 123;
  DA_send(x, &i, 1);
}
end

program Q
buffers
buffer
  int buf1
data_arrival_code
  incoming_data("buf1", DA_remove(buf1));
end

buffer
  int buf2
data_arrival_code
  incoming_data("buf2", DA_remove(buf2));
end
code
#define get_int(array, index) {((int *)array)[index]}
int buf1_tally, buf2_tally;
main()
{  buf1_tally = buf2_tally = 0;
}

incoming_data(char *name, struct buffer data *element)
{  if(strcmp(name, "buf1") == 0)
    buf1_tally++;
  else if(strcmp(name, "buf2") == 0)
    buf2_tally++;
  printf("%s, %d\n", name, get_int(element->array, 0));
  DA_free(element);
  if(buf1_tally >= 1 && buf2_tally >= 1)
    DA_exitapp(0);
}
end
```
program
R
outputs
output
   int y
end
code
main()
{|  int i = 456;
   DA_send(y, &i, 1);
|
end

H.6.3 Application Output

dash>>addhost smaug eldacar luthin elwing
dash>>addapp Test_6.dal
dash>>addprog Test_6.dapl Test_6
dash>>compile Test_6
dash>>createapp Test_6
dash>>buf1, 123
buf2, 456
status 1
Test_6 <id=1, RUNNING>
P <host=smaug, pid=22988, port=6002, exit_status=0, RUNNING> smaug
   x port=6003 (open, open)
Q <host=eldacar, pid=2867, port=6002, exit_status=0, RUNNING> eldacar
R <host=luthin, pid=6783, port=6002, exit_status=0, RUNNING> luthin
   y port=6004 (open, open)
proc 'P' of app 'Test_6, 1' terminated
proc 'Q' of app 'Test_6, 1' terminated
proc 'R' of app 'Test_6, 1' terminated
app 'Test 6, 1' TERMINATED
H.7 Test 7: Sending Arrays of Integers

This test is described in Section 8.2.7 and requires one DAI specification and two DAPL programs. These specification, programs, and application output are listed in the following subsections.

H.7.1 DAL Specifications

application Test 7
processes
  process P
  machines smouq
  outputs int x:buf:Q
end

process Q
  machines eldacar
  buffers int buf
end

H.7.2 DAPL Programs

program P
  outputs
    output
      int x
    end
  code
    #define SIZE 100

main()
{
  int data[SIZE], i;

  for(i = 0; i < SIZE; i++)
    data[i] = i * i;

  DA_send(x, data, 10);
  DA_send(x, &data[10], 5);
  DA_send(x, data, 100);
}
end
program;
buffers
buffer
    int buf

data_arrival_code
    struct buffer data *element;
    int *data, size, i;
    static int tally = 0;

element = DA_remove(buf);
data = element->array;
size = element->num_elements;

printf("Contents of array \&d: ", ++tally);
for(i = 0; i < size; i++)
    printf(" \&d", data[i]);
printf("\n");

DA_free(element);
if(tally >= 3)
    DA_exitapp(0);
end

H.7.3 Application Output

dash>>addhost smaug oldacar luthin olwing
dash>>addapp Test_7,dal
dash>>addprog Test_7,dapl Test_7
dash>>compile Test_7
dash>>createapp Test_7
dash>>Contents of array 1: 0 1 4 9 16 25 36 49 64 81
Contents of array 2: 100 121 144 169 196
Contents of array 3: 0 1 4 9 16 25 36 49 64 81 100 121 144 169 196 225 256 289 324 361 400 441 484 529 576 625 676 729 784 841 900 961 1024 1089 1156 1225 1296 1369 1444 1521 1600 1681 1764 1849 1936 2025 2116 2209 2304 2401 2500 2601 2704 2809 2916 3025 3136 3249 3364 3481 3600 3721 3844 3969 4096 4225 4356 4489 4624 4761 4900 5041 5184 5329 5476 5625 5776 5929 6084 6241 6400 6561 6724 6889 7056 7225 7396 7569 7744 7921 8100 8281 8464 8649 8836 9025 9216 9409 9604 9801
status 1
Test_7 <id=1, RUNNING>
> <host=smaug, pid=23183, port=6002, exit_status=0,
  RUNNING> smaug
> x port=6003 (open, open)
H.8 Test 8: Sending Arrays of Integers and Strings

This test is described in Section 8.2.8 and requires one DAL specification and two DAPL programs. These specification, programs, and application output are listed in the following subsections.

H.8.1 DAL Specifications

```
application Test_8
processes
  process P
    machines smauq
    outputs int x:buf1:0;
    char y:buf2:0
  end

  process Q
    machines eldacar
    buffers int buf1;
    char buf2
  end
end
```

H.8.2 DAPL Programs

```
program P
outputs
  output
    int x
  end
  output
    char y
  end
code
main()
{
  int a[] = {1, 2, 3, 4, 5};
  int b[] = {123, 456, 789};
  int c[] = {9, 8, 7, 6, 5, 4, 3, 2, 1, 0};
  char DAS[] = "Distributed Application System (DAS)";
```
char DAL[] = "DAL Language";
char DAPL[] = "DAPL Language";
char END[] = "END";

DA_send(y, DAS, strlen(DAS) + 1); // send NULL too
DA_send(x, a, 5);
DA_send(y, DAL, strlen(DAL) + 1);
DA_send(x, b, 3);
DA_send(y, DAPL, strlen(DAPL) + 1);
DA_send(x, c, 10);
DA_send(y, END, strlen(END) + 1);
}
end

program Q
buffers

buffer
int buf1

data_arrival_code
struct buffer_data *element;
int *data, 1;

element = DA_remove(buf1);
data = element->array;
for(i = 0; i < element->num_elements; i++)
    printf("%d ", data[i]);
printf("\n");
DA_free(element);
end

buffer
char buf2

data_arrival_code
struct buffer_data *element;

element = DA_remove(buf2);
printf("%s\n", element->array);
if(strcmp(element->array, "END") == 0)
{
    DA_free(element);
    DA_exit_app(0);
}
else
    DA_free(element);
end
H.8.3 Application Output

dash>>addhost smaug eldacar luthin elwing
dash>>addapp Test_8 dal
dash>>adddprog Test_8.dapl Test_8
dash>>compile Test_8
dash>>createapp Test_8
dash>>Distributed Application System (DAS) 1 2 3 4 5
DAL Language 123 456 789
DAPL Language 9 8 7 6 5 4 3 2 1 0
END
status 1
Test_8 <id=1, RUNNING>
P <host=smaug, pid=23237, port=6002, exit_status=0,
RUNNING> smaug
x port=6003 (open, open)
y port=6003 (open, open)
Q <host eldacar, pid=2956, port=6002, exit_status=0,
RUNNING> eldacar
proc 'P' of app 'Test_8', 1' terminated
proc 'Q' of app 'Test_8', 1' terminated
app 'Test_8', 1' TERMINATED

H.9 Test 9: Sending Arrays of Structures

This test is described in Section 8.2.9 and requires one DAL specification and two DAPL programs. These specification, programs, and application output are listed in the following subsections.

H.9.1 DAL Specifications

application Test_9
processes
    process P
    machines smaug
outputs struct test x:buf:Q
end

process Q
    machines eldacar
buffers struct test buf
end
end
H.9.2 DAPL Programs

```dapl
program P
structures
    struct test
    {
        int id;
        char name[10];
    };
outputs
    output struct test x
end
code
    main()
    {
        struct test a[] = { {1, "Bob"},
                           {2, "Kim"},
                           {3, "Sue"} };
        struct test b[] = { {4, "Angela"},
                           {5, "Peter"} };
        struct test c[] = { {6, "Jane"},
                           {7, "Paul"},
                           {8, "Alan"},
                           {9, "Amy"} };

        DA_send(x, a, 3);
        DA_send(x, b, 2);
        DA_send(x, c, 4);
    }
end

program Q
    structures
        struct test
        {
            int id;
            char name[10];
        };
    buffers
        buffer
            struct test buf
```
data_arrival_code
    struct buffer_data *element;
    struct test *data;
    int size, i;
    static int tally = 0;

    element = DA_remove(buf);
    data = element->array;
    size = element->num elements;

    printf("Contents of array %d: ", tally);
    for(i = 0; i < size; i++)
        printf(" (%d, %s)", data[i].id, data[i].name);
    printf("\n");

    DA_free(element);
    if(tally >= 3)
        DA_exitepp(0);

end

H.9.3 Application Output

dash:/>addhost smaug eldcar luthin elwing
dash:/>addapp Test_9.dal
dash:/>addprog Test_9.dal Test_9
dash:/>compile Test_9
dash:/>createapp Test_9
dash:/>Contents of array 1: (1, Hob) (2, Kim) (3, Sue)
Contents of array 2: (4, Angela) (5, Peter)
Contents of array 3: (6, Janc) (7, Paul) (8, Alan) (9, Amy)
status 1
Test_9 <id=1, RUNNING>
P <host=smaug, pid=23394, port=6002, exit_status=0,
    RUNNING> smaug
    x port=6003 (open, open)
Q <host-eldcar, pid 2977, port=6002, exit_status=0,
    RUNNING> eldcar
proc 'P' of app 'Test_9, 1' terminated
proc 'Q' of app 'Test_9, 1' terminated
app 'Test_9, 1' TERMINATED
Appendix I  DAL Specifications and DAPL Programs: Testing Hierarchy

Testing fundamental hierarchical DAL specifications is presented in Section 8.3 Testing Fundamental Hierarchies. The DAL specifications, DAPL programs and application outputs of such test applications are presented in this appendix.

I.1  Test 1: Parent and Child Relationship

This test is described in Section 8.3.1 and requires two DAL specification and three DAPL programs. These specification, programs, and application output are listed in the following subsections.

I.1.1  DAL Specifications

```plaintext
application  Fibonacci_sequences
machines     smaug, eldacar, luthin
creation     display_sequence
processes    
  process     display_sequence
  buffers     int sequence
  outputs     int N: length: Fibonacci_sequence
end

process     Fibonacci_sequence
requires    fibonacci
buffers     int length
outputs     int seq: sequence: display_sequence
end

application  Fibonacci
machines     smaug, eldacar, luthin
buffers      int length:create_sequence
outputs      int seq:create_sequence
creation     create_sequence
processes    
  process     create_sequence
  machines   smaug, eldacar
  buffers    int length;
              int buf
```
outputs  int a: buf: compute_Fibonacci
         int seq:
end

process  compute_Fibonacci
machines luthin, eldcar
buffers  int buf
outputs  int b: buf: create_sequence
end

I.1.2 DAPL Programs

program  display_sequence
buffers  
buffer
         int sequence
data_arrival_code
         #define get_int(array, index) (*((int *)array)[index])
         struct buffer_data *element;
         int i;

         element = DA_remove(sequence);
         for(i = 0; i < element->num_elements; ++i)
         {
             printf("%d", get_int(element->array, i));
             printf("\n");
             DA_free(element);
         }
         if(n < SIZE)
         {
             DA_send(N, &sequence_length[n++], 1);
         }
         else
         {
             DA_exitapp(0);
         }
end
outputs  
output
         int N
end
code

#define SIZE 3
int sequence_length[SIZE] = {5, 7, 10};
int n = 0;

main()
{
    DA_send(N, &sequence_length[n++], 1);
}
end
program
  create sequence
buffers
  buffer
    int length
  data_arrival_code
    struct buffer_data *element;

    element = DA_remove(length);
    N = get_int(element->array, 0);
    Fibonacci = (int *)malloc(N * sizeof(int));
    DA_free(element);

    Fibonacci[0] = 1;
    Fibonacci[1] = 1;
    next = 2;
    DA_send(a, Fibonacci, 2);
  end

buffer
  int buf
  data_arrival_code
    struct buffer_data *element;

    element = DA_remove(buf);
    Fibonacci[next++] = get_int(element->array, 0);
    DA_free(element);

    if(next < N)
      DA_send(a, &Fibonacci[next-2], 2);
    else
      { DA_send(seq, Fibonacci, N);
        free(Fibonacci);
      }
  end
outputs
  output
    int a
  end
output
  int seq
end
Appendix I. DAL Specifications and DAPL Programs: Testing Hierarchy

```c
#include stdsym

int *Fibonacci, N, next;

main()
{
}
end

program compute_Fibonacci
buffers
buffer
int buf

data_arrival_code
#define get_int(array, index) (*((int *)array)[index])
struct buffer data *element;
int next_Fibonacci;

element = DA remove(buf);
next_Fibonacci = get_int(element->array, 0) +
    get_int(element->array, 1);
DA free(element);
DA send(b, &next_Fibonacci, 1);
end
outputs
output
int b
end

1.1.3 Application Output

dash>> addhost smaug eldacar luthin elwing
dash>> addapp Fibonacci_sequences.dal
dash>> addprog Fibonacci_sequences.dap1 Fibonacci_sequences
dash>> addapp Fibonacci.dal
dash>> addprog Fibonacci.dapl Fibonacci
dash>> compile Fibonacci_sequences
dash>> createapp Fibonacci_sequences
dash>> l 1 1 2 3 5
    1 1 2 3 5 8 13
    1 1 2 3 5 8 13 21 34 55
status 1
Fibonacci sequences <id=1, RUNNING> smaug eldacar luthin
compute_Fibonacci
  <host=luthin, pid=/188, port=6002, exit_status=0,
RUNNING> luthin eldacar
  b port=6004 (open, open)
create_sequence
  <host=smaug, pid=23479, port=6003, exit_status=0,
RUNNING> smaug eldacar
  a port=32979 (open, open)
  seq port=6005 (open, open)
display_sequence
  <host=smaug, pid=23485, port=6003, exit_status=0,
RUNNING> smaug eldacar luthin
  N port=57184 (open, open)
proc 'compute_Fibonacci' of app 'Fibonacci_sequences, 1'
  terminated
proc 'create_sequence' of app 'Fibonacci_sequences, 1'
  terminated
proc 'display_sequence' of app 'Fibonacci_sequences, 1'
  terminated
app 'Fibonacci_sequences, 1' TERMINATED

I.2  Test 2: Parent, Child and Grandchild Relationships

This test is described in Section 8.3.2 and requires three DAL specification and four DAPL programs. These specification, programs, and application output are listed in the following subsections.

I.2.1  DAL Specifications

application          sequences
machines             smaug, eldacar, luthin, elwing
creation             sequence_length
processes
    process        sequence_length
    outputs        int lengths: lengths: display_sequences
    end

    process        display_sequences
    requires       Fibonacci_sequences
    buffers        int lengths
    end

end
application Fibonacci_sequences
machines smaug, eldacar, luthin
buffers int lengths; display_sequence
creation display_sequence
processes
  process display_sequence
  buffers int lengths;
      int sequence
  outputs int N: length: Fibonacci_sequence
end

process Fibonacci_sequence
  requires Fibonacci
  buffers int length
  outputs int seq: sequence: display_sequence
end
end

application Fibonacci
machines smaug, eldacar, luthin
buffers int length:create_sequence
outputs int seq:create_sequence
creation create_sequence
processes
  process create_sequence
  machines smaug, eldacar
  buffers int length;
      int buf
  outputs int a: buf: compute_Fibonacci;
      int seq:
end

process compute_Fibonacci
  machines luthin, eldacar
  buffers int buf
  outputs int b: buf: create_sequence
end
end

1.2.2 DAPL Programs

program sequence_length
outputs
  output
    int lengths
end
code
main()
{
    int sequence_lengths[] = {5, 7, 10};
    DA_send(lengths, sequence_lengths, 3);
}
end

program
display_sequence
buffers
buffer
    int lengths
data_arrival_code
    sequence_data = DA_remove(lengths);
    sequence_lengths = sequence_data->array;
    size = sequence_data->num_elements;
    DA_send(N, &sequence_lengths[n++], 1);
end
buffer
    int sequence
data_arrival_code
    #define get_int(array, index) {((int *)array)[index]}
    struct buffer_data *element;
    int i;

    element = DA_remove(sequence);
    for(i = 0; i < element->num_elements; i++)
        printf("%d ", get_int(element->array, i));
    printf("\n");
    DA_free(element);

    if(n < size)
        DA_send(N, &sequence_lengths[n++], 1);
    else
        {
            DA_free(sequence_data);
            DA_exitapp(0);
        }
end
outputs
output
    int N
end


```c
struct buffer_data *sequence_data;
int *sequence_lengths;
int size;
int n = 0;

main()
{
}
}
end

program
create sequence
buffers
buffer
int length

data_arrival_code
struct buffer_data *element;

element = DA_remove(length);
N = get_int(element->array, 0);
Fibonacci = (int *)malloc(N * sizeof(int));
DA_free(element);

Fibonacci[0] = 1;
Fibonacci[1] = 1;
next = 2;
DA_send(a, Fibonacci, 2);
end

buffer
int buf

data_arrival_code
struct buffer_data *element;

element = DA_remove(buf);
Fibonacci[next++] = get_int(element->array, 0);
DA_free(element);

if(next < N)
    DA_send(a, &Fibonacci[next-2], 2);
else
{
    DA_send(seq, Fibonacci, N);
    free(Fibonacci);
}
end
```
outputs
  output
    int a
  end
output
  int seq
end
code
  #define get_int(array, index) (((int*)array)[index])
  int *Fibonacci, N, next;

main()
{
}
end

program
  compute_Fibonacci
buffers
  buffer
    int buf
  data_arrival_code
    #define get_int(array, index) (((int*)array)[index])
    struct buffer_data *element;
    int next_Fibonacci;

    element DA_remove(buf);
    next_Fibonacci - get_int(element->array, 0) +
    get int(element->array, 1);
    DA_free(element);
    DA_send(b, &next_Fibonacci, l);
  end
outputs
  output
    int b
  end
I.2.3 Application Output

dash>> addhost smaug eldacar luthin elwing
dash>> addapp sequences.dal
dash>> addprog sequences.dap1 sequences
dash>> addapp Fibonacci_sequences.dal
dash>> addprog Fibonacci_sequences.dap1 Fibonacci_sequences
dash>> addapp Fibonacci.dal
dash>> addprog Fibonacci.dap1 Fibonacci
dash>> compile sequences
dash>> createapp sequences

dash>> 1 1 2 3 5
1 1 2 3 5 8 13
1 1 2 3 5 8 13 21 34 55

status 1

sequences <id=1, RUNNING> smaug eldacar luthin elwing
compute_Fibonacci
  <host=luthin, pid=/250, port=6002, exit_status=0, RUNNING> luthin eldacar
  b port=6005 (open, open)
create_sequence
  <host=smaug, pid=23503, port=6002, exit_status=0, RUNNING> smaug eldacar
    a port=32990 (open, open)
    seq port=6006 (open, open)
display_sequence
  <host=smaug, pid=23504, port=6003, exit_status=0, RUNNING> smaug eldacar luthin
    N port=57294 (open, open)
sequence_length
  <host=smaug, pid=23606, port=6004, exit_status=0, RUNNING> smaug eldacar luthin elwing
    length 6007 (open, open)
proc 'compute_Fibonacci' of app 'sequences', 1' terminated
proc 'create_sequence' of app 'sequences', 1' terminated
proc 'display_sequence' of app 'sequences', 1' terminated
proc 'sequence_length' of app 'sequences', 1' terminated
app 'sequences', 1' TERMINATED
1.3 Test 3: Parent of Several Children

This test is described in Section 8.3.3 and requires three DAL specification and three DAPL programs. These specification, programs, and application output are listed in the following subsections.

1.3.1 DAL Specifications

```plaintext
application  Fibonacci_sequences
machines     smaug, eldacar, luthin, c1wing
creation      display_sequence
processes
  process  display_sequence
  requires sequences
  buffers  int sequence
  outputs  int N: length: Fibonacci_sequence
end

  process  Fibonacci_sequence
  requires Fibonacci
  buffers  int length
  outputs  int seq: sequence: display_sequence
end
end

application  sequences
machines     smaug, eldacar, luthin
buffers      int sequence:display_sequence
outputs      int N:display_sequence
creation     sequence length
processes
  process  sequence_length
  outputs  int length: lengths: display_sequence
end

  process  display_sequence
  buffers  int lengths;
  int sequence
  outputs  int N:
end
end
```
Appendix I. DAL Specifications and DAPL Programs: Testing Hierarchy

```
application Fibonacci
machines smaug, eldacar, luthin
buffers int length; create_sequence
outputs int seq; create_sequence
creation create_sequence
processes
  process create_sequence
    machines smaug, eldacar
    buffers int length;
    int buf
    outputs int a; buf; compute_Fibonacci;
    int seq:
    end

  process compute_Fibonacci
    machines luthin, eldacar
    buffers int buf
    outputs int b; buf; create_sequence
    end
end

I.3.2 DAPL Programs

program sequence_length
  outputs
    output
    int lengths
  end
  code
    main()
    { int sequence_lengths[] = {5, 7, 10};
      DA_send(lengths, sequence_lengths, 3);
    }
end

program display_sequence
buffers
  buffer
    int lengths
  data_arrival_code
    sequence_data = DA_remove(lengths);
    sequence_lengths = sequence_data->array;
    size = sequence_data->num_elements;
    DA_send(N, &sequence_lengths[n+1], 1);
end
```
buffer
  int sequence

data_arrival_code
  #define get_int(array, index) (((int *)array)[index])
  struct buffer_data *element;
  int i;

  element DA_remove(sequence);
  for(i = 0; i < element->num_elements; i++)
    printf("%d ", get_int(element->array, i));
  printf("\n");
  DA_free(element);

  if(n < size)
    DA_send(N, sequence_lengths[n++], 1);
  else
  {
    DA_free(sequence_data);
    DA_exitapp(0);
  }
end
outputs
  output
    int N
end

code
  struct buffer_data *sequence_data;
  int *sequence_lengths;
  int size;
  int n 0;

main()
{
}
end

program
  create sequence
buffers
  buffer
    int length
  data_arrival_code
    struct buffer_data *element;
element = DA_remove(length);
N = get_int(element->array, 0);
Fibonacci = (int *)malloc(N * sizeof(int));
DA_free(element);

Fibonacci[0] = 1;
Fibonacci[1] = 1;
next = 2;
DA_send(s, Fibonacci, 2);
end

buffer
int buf
data_arrival_code
struct buffer_data *element;

element = DA_remove(buf);
Fibonacci[next++] = get_int(element->array, 0);
DA_free(element);

if(next < N)
    DA_send(s, &Fibonacci[next-2], ?);
else
    { 
        DA_send(seq, Fibonacci, N);
        free(Fibonacci);
    }
end
outputs
output
  int a
end
output
  int seq
end
code
#define get_int(array, index) {{{(int *)array}[index]}}
int *Fibonacci, N, next;

main()
{
}
end
program
  compute_Fibonacci
buffers
  buffer
    int buf
data_arrival_code
  #define get_int(array, index) (((int *)array)[index])
  struct buffer_data *element;
  int next_Fibonacci;

  element = DA_remove(buf);
  next_Fibonacci = get_int(element->array, 0) +
    get_int(element->array, 1);
  DA_free(element);
  DA_send(b, &next_Fibonacci, 1);
end
outputs
  output
    int b
end

I.3.3 Application Output

dash>>addhost smaug eldacar luthin elwing
dash>>addapp Fibonacci_sequences.dal
dash>>addapp sequences.dal
dash>>addprog sequences.dapl sequences
dash>>addapp Fibonacci.dal
dash>>addprog Fibonacci.dapl Fibonacci
dash>>compile Fibonacci_sequences
dash>>createapp Fibonacci_sequences
dash>>1 1 2 3 5
  1 1 2 3 5 8 13
  1 1 2 3 5 8 13 21 34 55
status 1
Fibonacci_sequences
  <id=1, RUNNING> smaug eldacar luthin elwing
compute_Fibonacci
  <host luthin, pid=7298, port=6002, exit_status=0, RUNNING> luthin cd/docar
  b port=6005 (open, open)
create_sequence
  <host=smaug, pid=23798, port=6003, exit_status=0, RUNNING> smaug eldacar
  a port=33001 (open, open)
  seq port=6006 (open, open)
I.4 Test 4: Single DAL Specification for Several Children

This test is described in Section 8.3.4 and requires two DAL specification and three DAPL programs. These specification, programs, and application output are listed in the following subsections.

I.4.1 DAL Specifications

```
application Fibonacci_sequences
machines smaug, eldacar, luthin
creation display_sequence
processes
  process display_sequence
  machines eldacar
  buffers int sequence
  outputs int N_1: length: seq_1;
     int N_2: length: seq_2;
     int N_3: length: seq 3
  end

  process seq_1
  requires Fibonacci
  buffers int length
  outputs int seq: sequence: display sequence
  end
```
process seq 2
requires Fibonacci
buffers int length
outputs int seq; sequence: display_sequence
end

process seq 3
requires Fibonacci
buffers int length
outputs int seq; sequence: display_sequence
end
end

application Fibonacci
machines smaug, eldacar, luthin
buffers int length; create_sequence
outputs int seq; create_sequence
creation create_sequence
processes
process create_sequence
machines smaug, eldacar
buffers int length;
   int buf
outputs int a; buf; compute_Fibonacci;
   int seq;
end

process compute_Fibonacci
machines luthin, eldacar
buffers int buf
outputs int b; buf; create_sequence
end
end

1.4.2 DAPL Programs

program
   display_sequence
buffers
   buffer
      int sequence
   data_arrival_code
#define get_int(array, index) (((int *)array)[index])
struct buffer_data *element;
int i;
static int tally = 0;
element = DA_remove(sequence);
for (i = 0; i < element->num_elements; i++)
    printf("%d ", got_int(element->array, i));
printf("\n");
DA_free(element);

if (++tally >= 3)
    DA_exitapp(0);
end
outputs
output
    int N_1
end
output
    int N_2
end
output
    int N_3
end
code
main()
{
    int sequence_lengths[] = {5, 7, 10};
    DA_send(N_1, &sequence_lengths[0], 1);
    DA_send(N_2, &sequence_lengths[1], 1);
    DA_send(N_3, &sequence_lengths[2], 1);
}
end

program
    create sequence
buffers
    buffer
        int length
    data_arrival_code
        struct buffer_data *element;
        element = DA_remove(length);
        N = got_int(element->array, 0);
        Fibonacci = (int *)malloc(N * sizeof(int));
        DA_free(element);
Fibonacci[0] - 1;
Fibonacci[1] = 1;
next -= 2;
 DA_send(a, Fibonacci, 2);

end

buffer
int buf

data_arrival_code
struct buffer_data *element;

    element = DA_remove(buf);
    Fibonacci[next++] = get_int(element->array, 0);
    DA_free(element);

    if(next < N)
        DA_send(a, &Fibonacci[next-2], 2);
    else
        {
            DA_send(seq, Fibonacci, N);
            free(Fibonacci);
        }

end

outputs
output
int a
end

output
int seq
end

code
#define get_int(array, index) (*((int *)array)[index])
int *Fibonacci, N, next;

main()
{
}
end
program
  compute_Fibonacci
buffers
  buffer
    int buf
  data_arrival_code
    #define get_int(array, index) {((int *)array)[index]}
    struct buffer_data *element;
    int next_Fibonacci;
    element = DA_remove(buf);
    next_Fibonacci = get_int(element->array, 0) +
        get_int(element->array, 1);
    DA_free(element);
    DA_send(b, &next_Fibonacci, 1);
end
outputs
  output
    int b
end

1.4.3 Application Output

dash>>addhost smaug eldacar luthin elwing
dash>>addapp Fibonacci_sequences.dal
dash>>addprog Fibonacci_sequences.dapl Fibonacci_sequences
dash>>addapp Fibonacci.dal
dash>>addprog Fibonacci.dapl Fibonacci
dash>>compile Fibonacci_sequences
dash>>createapp Fibonacci_sequences

dash>>1 1 2 3 5
1 1 2 3 5 8 13
1 1 2 3 5 8 13 21 34 55

status 1
Fibonacci_sequences
  <id=1, RUNNING> smaug eldacar luthin
  compute_Fibonacci <host=luthin, pid=7409, port=6003, exit_status=0, RUNNING> luthin eldacar
b port=6005 (open, open)
create_sequence <host=smaug, pid=24574, port=6002, exit_status=0, RUNNING> smaug eldacar
  a port=33024 (open, open)
  seq port=6003 (open, open)
compute_Fibonacci <host=luthin, pid=7408, port=6002, exit_status=0, RUNNING> luthin eldacar
b port=6006 (open, open)
create_sequence <host=smaug, pid=24591, port=6003, exit_status=0, RUNNING> smaug eldacar
  a port=33027 (open, open)
  seq port=6004 (open, open)
compute_Fibonacci <host=luthin, pid=7427, port=6004, exit_status=0, RUNNING> luthin eldacar
  b port=6007 (open, open)
create_sequence <host=smaug, pid=24614, port=6004, exit_status=0, RUNNING> smaug eldacar
  a port=33030 (open, open)
  seq port=6005 (open, open)
display_sequence <host=eldacar, pid=3015, port=6002, exit_status=0, RUNNING> eldacar
  N 1 port=60495 (open, open)
  N 2 port=60507 (open, open)
  N 3 port=60519 (open, open)
proc 'compute_Fibonacci'
  of app 'Fibonacci_sequences, 1' terminated
proc 'create_sequence'
  of app 'Fibonacci_sequences, 1' terminated
proc 'compute_Fibonacci'
  of app 'Fibonacci_sequences, 1' terminated
proc 'create_sequence'
  of app 'Fibonacci_sequences, 1' terminated
proc 'compute_Fibonacci'
  of app 'Fibonacci_sequences, 1' terminated
proc 'create_sequence'
  of app 'Fibonacci_sequences, 1' terminated
proc 'display_sequence'
  of app 'Fibonacci_sequences, 1' terminated
app 'Fibonacci_sequences, 1' TERMINATED
Appendix J  DAS Generated C Code:
Example

The distributed application to compute and display Fibonacci numbers presented in
Section 4.5 Example: Distributed Fibonacci is specified using the DAL specifications
and DAPI programs shown in Figure 4.17, Figure 4.18 and Figure 4.19. The C code
generated by the DAS system from the specification and programs is presented in this
appendix. Two C programs are presented: one corresponding to the process called
display_fibonacci, the other corresponding to process compute_fibonacci.

Generated C Code for Process display_fibonacci.
/*
   C program generated by DAS.
   DAPI program name is 'display_fibonacci'.
*/
#include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <strings.h>
#include <flow.h>
#include <logical_flow.h>
#include <buffer.h>
#include <idp.h>
#include <type_id.h>
#include <das_states.h>
#include <disposition.h>
#include <server_sd.h>

/* Those two values form the unique identifier */
/* of this process with respect to the das system. */
unsigned long das_app_id;
char *das_process_name;

/* Those two values form the unique identifier */
/* of the well known port of the das system. */
char *das_hostname;
int das_portnum;

/* The socket descriptor and port number used by */
/* by this process to receive messages from the */
/* das system. */
int server_sd;
int server_portnum;
/* The state of the process. */
int das_process_state = UNDEFINED;

/* these array values come from preprocessing */
struct flow_data flow_data[] =
{
    "o0_1_0", "pi_0", 0, 0, 0, NULL,
    "o0_0_0", "pi_0", 0, 0, 0, NULL
};

/* Outputs. */
OUTPUT *a; /* o0_0_0 */

/* DAPL buffers. */
BUFFER *buf;

BUFFER_TABLE *buffer_table;

/* DAPL structures */
/* no structures specified */

/* prototypes */
int program_main();
int program_manager(struct buf table *buffer table);
void pre_move();
void post_move();
void buf data arrival();
void buf_pre_remove();
void buf_post_remove();
void e_pre_send();
void a_post_send();

main(int argc, char **argv)
{
    pid_t pid;
    char data[BUFSIZE];
    int exit status;

    if(argc != 7)
    {
        printf("das system error: ");
        printf("cannot initialise identifiers for 'n'\n", argv[0]);
        DAS_exit(1);
    }

    /* Set the signal dispositions for this process. */
    proc_dispositions();
}
/* Initialise process identifiers. */
das_app_id = atoi(argv[1]);
das_process_name = strdup(argv[2]);

/* Initialise das identifiers. */
das_hostname = strdup(argv[3]);
das_portnum = atoi(argv[4]);

/* Initialise process state. */
das_process_state = atoi(argv[5]);

/* Initialise the last port number used. */
last_portnum_used = das_portnum;

/* Create the server socket for this process. */
server_portnum = last_portnum_used + 1;
server_sd = create_server_socket(&server_portnum);
last_portnum_used = server_portnum;

/* send the process's server port number back to das */
pid = getpid();
printf(das_hostname, "%d %ld", server_portnum, pid);
send_das_message(das_hostname, das_process_name,
    DAS_ID, DAS_NAME,
    das_app_id, das_process_name,
    PROCESS_SERVER_PORT, data);

switch(das_process_state)
{
    case CREATION:
        install_logical_flows(2); /* das_process_state is READY at this point */
        create_outputs();
        create_buffers();
        while(das_process_state != RUNNING)
            wait_for_das();
        program_main();
        break;

    case CREATION_V2:
        install_no_logical_flows(2); /* das_process_state is READY at this point */
        create_outputs();
        create_buffers();
/* Inform das that the process is running. */
send_das_message(das_hostname, das_portnum,
    DAS_ID, DAS_NAME,
    das_app_id, das_process_name,
    PROCESS_RUNNING, NULL);

program_main();
break;

case MOVING:
    install_logical_flows(2);
    /* das process state is the pre move state at this point */
    das_process_state = atoi(argv[6]);

    create_outputs();
    create_buffers();

    /* Inform das that the process is running. */
    send_das_message(das_hostname, das_portnum,
        DAS_ID, DAS_NAME,
        das_app_id, das_process_name,
        PROCESS_RUNNING, NULL);
    post_move();
    break;
}

exit_status = program_manager(buffer_table);
DA_exit(exit_status);
}

int program_manager(struct buf_table *buffer_table)
{
    int nbytet, n, i, j, size, flow_id_set[BUFSIZ];
    int processing, exit_status;
    struct bdp idp;
    struct buffer_data *data;
    BUFFER *buf;
    char tcmp_data[BUFSIZ];

    exit_status = 0;
    processing = 1;
    while(processing)
    {
        fflush(stdout);
        n = wait for input(flow id set);
        for(i = 0; i < n; i++, flush(stdout))
        {
if(flow_id_set[i] == server_sd)
{
    process_das_message();
    continue;
}

if((nbytes = receive_IDP(flow_id_set[i], &IDP)) == 0)
{
    /* Normally, this should not happen. */

    /* Socket source is closed. */
    /* Source process has probably exited unexpectedly. */
    /* Need to clean up. */
    /* Close all logical flows on this socket. */

    for(j = 0; j < num_logical_flows; j++)
    {
        if(flow_id_set[i] == flow_id[flow_data[j].internal_fid])
        {
            if(strcmp(flow_data[j].source_process_name,
                       das_process_name) == 0)
            {
                close_channel_source(flow_data[j].flow_name,
                                      flow_data[j].source_process_name);
                sprintf(tmp_data, "%s", flow_data[j].flow_name);
                send_das_message(das_hostname, das_portnum,
                                 DAS_ID, DAS_NAME,
                                 das_app_id, das_process_name,
                                 PROCESS_DEST_PROC_MISSING,
                                 temp_data);
            }
            else
            {
                close_channel_dest(flow_data[j].flow_name,
                                    flow_data[j].source_process_name);
                sprintf(tmp_data, "%s", flow_data[j].source_process_name);
                send_das_message(das_hostname, das_portnum,
                                 DAS_ID, DAS_NAME,
                                 das_app_id, das_process_name,
                                 PROCESS_SOURCE_PROC_MISSING,
                                 temp_data);
            }
        }
    }

    continue;
}
data = (struct buffer_data *)
    malloc(sizeof(struct buffer_data));
data->num_type_info_elements =
    IDP.header.num_type_info_elements;
size = sizeof(struct type_info) *
    data->num_type_info_elements;
memcp (data->type_info, IDP.type_info, size);

data->element_size = IDP.header.element_size;
data->num_elements = IDP.header.num_elements;

size = data->element_size * data->num_elements;
data->array = (void *)malloc(size);
memcp (data->array, IDP.data, size);

buf = BT_buffer(buffer_table, IDP.header.buffer_id);
it(but)
{
    insert(buf, data);
    buf->arrival();
}
}

flush(stdout);
return(exit_status);

#define main program_main

/* user code */

#define got_int(array, in dex) {(*(int *)array)[index]}

int tally = 0;
int array[2];
int prev_fibonacci = 1;
int next_fibonacci = 1;

main()
{
    printf("Fib(%d) = %d
", tally, prev_fibonacci);
    printf("Fib(%d) = %d
", tally, next_fibonacci);
    DA_send(a, array, 2);
}
/* pre move code */
void pre_move()
{
    /* no code specified */
}


/* post move code */
void post_move()
{
    /* no code specified */
}

/* buffer functions */
void buf data arrival()
{
    /* Source file = /home/rad/DA/examples/hierarchy/exitapp test/
        fibonacci/Fibonacci.dapl */
    /* DAPL buffer name = buf */
    /* offset = 70, length = 281 */

    struct buffer_data *element;

    element = DA_remove(buf);
    prev_Fibonacci = next_Fibonacci;
    next_Fibonacci = get_int(element->array, 0);
    DA_free(element);

    printf("fib(%d) = %d\n", i+tally, next_Fibonacci);
    if(tally < 10)
        DA_send(a, array, 2);
    else
        DA_exitapp(0);

}

void buf_pre_remove()
{
    /* no code specified */
}

void buf_post_remove()
{
    /* no code specified */
}
Appendix J: DAS Generated C Code: Example

/* output functions */
void a_pre_send()
{
    /* Source file ... */
    /* DAPL output name = a */
    /* offset = 393, length = 58 */

        array[0] = prev_Fibonacci;
        array[1] = next_Fibonacci;

}

void a_post_send()
{
    /* no code specified */
}

/* Generated output function. */
void create_outputs()
{
    int num_elements;

    a = create output();
    if(a->installed = get_installed(flow_data, "o0_0_0"))
        a->ad = flow id[ get fid index(flow data, "o0 0 0") ];
    insert_output(flow_data, "o0_0_0", a);
    a->bsize = sizeof(int);
    num_elements = 1;
    a->type_info = { struct type_info *
        malloc(sizeof(struct type_info) * num_elements);
    a->num_type_info_elements = num_elements;
    a->type_info[0].type_id = 3;
    a->type_info[0].struct_id = -1;
    a->type_info[0].type_size = 4;
    a->type_info[0].num_elements = 1;
    a->bid = 0;
    a->proc_send = a.proc_send;
    a->post_send = a.post_send;
}
/* Generated buffer function. */
void create_buffers()
{
    buffer_table = BT_create();
    BT_init(buffer_table);

    buf = create buffer();
    buf->arrival = (void *)buf data arrival;
    buf->pre_remove = (void *)buf_pre_remove;
    buf->post_remove = (void *)buf_post_remove;
    BT_insert(buffer_table, buf);
}
Generated C Code for Process compute_Fibonacci.

    /*
    * C program generate by DAS.
    * DAPL program name is 'compute_Fibonacci'.
    */
#define include <stdio.h>
#include <unistd.h>
#include <sys/types.h>
#include <strings.h>

#include <flow.h>
#include <logical_flow.h>
#include <buffer.h>
#include <idp.h>
#include <type.h>

#include <das_states.h>
#include <disposition.h>
#include <server_sd.h>

    /* These two values form the unique identifier */
    /* of this process with respect to the das system. */
    unsigned long das_app_id;
    char *das_process_name;

    /* These two values form the unique identifier */
    /* of the well known port of the das system. */
    char *das_hostname;
    int das_portnum;

    /* The socket descriptor and port number used by */
    /* by this process to receive messages from the */
    /* das system. */
    int server_sd;
    int server_portnum;

    /* The state of the process. */
    int das_process_state = UNDEFINED;

    /* these array values come from preprocessing */
    struct flow_data flow_data[] =
    {
    /* Outputs. */
    OUTPUT *b;}/#{0_1_0}*/
/* DAPI buffers. */
BUFFER *buf;

BUFFER_TABLE *buffer_table;

/* DAPI structures */
/* no structures specified */

/* prototypes */
int program_main();
int program_manager(struct buf_table *buffer_table);
void pre_move();
void post_move();
void buf_data_arrival();
void buf_pre_remove();
void buf_post_remove();
void b_pre_send();
void b_post_send();

main(int argc, char **argv)
{
    pid_t pid;
    char data[BUFSIZE];
    int exit_status;

    if(argc != 7)
    {
        printf("das system error: ");
        printf("cannot initialise identifiers for 's'\n", argv[0]);
        DA_exit(1);
    }

    /* Set the signal dispositions for this process. */
    proc_dispositions();

    /* Initialise process identifiers. */
    das_app_id = atol(argv[1]);
    das_process_name = strdup(argv[2]);

    /* Initialise das identifiers. */
    das_hostname = strdup(argv[3]);
    das_portnum = atoi(argv[4]);

    /* Initialise process state. */
    das_process_state = atoi(argv[5]);

    /* Initialise the last port number used. */
    last_portnum_used = das_portnum;
/* Create the server socket for this process. */
server_portnum = last_portnum_used + 1;
server_sd = create_server_socket(iserver_portnum);
last_portnum_used = server_portnum;

/* send the process's server port number back to das */
pid = getpid();
sprintf(data, "%d %ld", server_portnum, pid);
send_das_message(das_hostname, das_portnum,
    DAS_ID, DAS_NAME,
    das_app_id, das_process_name,
    PROCESS_SERVER_PORT, data);

switch(das_process_state)
{
    case CREATION:
        install_logical_flows(2);
        /* das_process_state is READY at this point */

        create_outputs();
        create_buffers();

        while(das_process_state != RUNNING)
        {
            wait_for_das();

            program_main();
            break;
        }

    case CREATION_V2:
        install_no_logical_flows(2);
        /* das_process_state is READY at this point */

        create_outputs();
        create_buffers();

        /* Inform das that the process is running. */
        send_das_message(das_hostname, das_portnum,
            DAS_ID, DAS_NAME,
            das_app_id, das_process_name,
            PROCESS_RUNNING, NULL);

        program_main();
        break;
case MOVING:
    install logical_flows(2);
    /* das_process state is the pre move state at this point */
    das_process_state = atoi(argv[6]);

    create_outputs();
    create_buffers();

    /* Inform das that the process is running. */
    send_das_message(das_hostname, das_portnum,
        DAS ID, DAS NAME,
        das_app_id, das_process_name,
        PROCESS RUNNING, NULL);
    post_move();
    break;
}

exit_status = program_manager(buffer_table);
DA_exit(exit_status);
}

int program_manager(struct buf_table *buffer_table)
{
    int nbytes, n, i, j, size, flow_id_set[BUFSIZE];
    int processing, exit_status;
    struct idp IDP;
    struct buffer data *data;
    BUFFER *buf;
    char temp data[BUFSIZE];

    exit status = 0;
    processing = 1;
    while (processing)
    {
        fflush(stdout);
        n = wait_for_input(flow_id_set);
        for (i = 0; i < n; i++, fflush(stdout))
        {
            if (flow_id_set[i] == server_sd)
            {
                process_das_message();
                continue;
            }
            if ((nbytes = receive_IDP(flow_id_set[i], &IDP)) > 0)
            {
                /* Normally, this should not happen. */

                /* Socket source is closed. */
/* Source process has probably exited unexpectedly. */
/* Need to clean up. */
/* Close all logical flows on this socket. */

for (i = 0; i < num_logical_flows; i++)
{
    if (flow_id_set[i] == flow data[j].internal fid)
    {
        if (strcmp(flow data[j].source_process_name,
                   das_process_name) == 0)
        {
            close_channel source(flow data[j].flow_name,
                                 flow data[j].source_process_name);
            sprintf(temp_data, "%s", flow data[j].flow_name);
            send_das_message(das_hostname, das_portnum,
                             DAS_ID, DAS_NAME,
                             das_app_id, das_process_name,
                             PROCESS_DEST_PROC_MISSING,
                             temp_data);
        }
        else
        {
            close_channel dest(flow data[j].flow_name,
                                flow data[j].source_process_name);
            sprintf(temp_data, "%s",
                    flow data[j].source_process_name);
            send_das_message(das_hostname, das_portnum,
                             DAS_ID, DAS_NAME,
                             das_app_id, das_process_name,
                             PROCESS_SOURCE_PROC_MISSING,
                             temp_data);
        }
    }
}

continue;

data = (struct buffer data *)
malloc(sizeof(struct buffer_data));
data->num_type_info_elements = IDP.header.num_type_info_elements;
size = sizeof(struct type_info) *
data->num_type_info_elements;
mncpy(data->type_info, IDP.type_info, size);
data->element size = IDP.header.element size;
data->num_elements = IDP.header.num_elements;
size = data->element_size * data->num_elements;
data->array = (void *)malloc(size);
memcpy(data->array, IDP.data, size);

buf = BT_buffer(buffer_table, IDP.header.buffer_id);
if(buf)
{
    insert(buf, data);
    buf->arrival();
}
}
}

fflush(stdout);
return(exit_status);
}

#define main program main

/* user code */
/* no code specified */
main()
{
}

/* pre move code */
void pre_move()
{
    /* no code specified */
}

/* post move code */
void post_move()
{
    /* no code specified */
}

/* buffer functions */
void buf_data_arrival()
{
    /* Source file */
    /* offset = 812, length = 271 */
#define get_int(array, index) (((int *)array)[index])
struct buffer_data *element;
int next_Fibonacci;

element = DA_remove(buf);
next_Fibonacci = get_int(element->array, 0) +
                get_int(element->array, 1);
DA_free(element);
DA_send(b, &next_Fibonacci, 1);
}

void buf_pre_remove()
{
    /* no code specified */
}

void buf_post_remove()
{
    /* no code specified */
}

/**< output functions */
void b_pre_send()
{
    /* no code specified */
}

void b_post_send()
{
    /* no code specified */
}

/**< Generated output function */
void create_outputs()
{
    int num_elements;

    b = create_output();
    if(b->installed = get_installed(flow_data, "c0_1_0"))
        b->sd = flow_id[ get_fd_index(flow_data, "c0_1_0" ) ];
    insert_output(flow_data, "c0_1_0", b);
    b->tsize = sizeof(int);
    num_elements = 1;
    b->type_info = (struct type_info *)
        malloc(sizeof(struct type_info *) * num_elements);
b->num_type_info_elements = num_elements;
b->type_info[0].type_id = 3;
b->type_info[0].struct_id = -1;
b->type_info[0].type_size = 4;
b->type_info[0].num_elements = 1;
b->bid = 0;
b->pre_send = b_pre_send;
b->post_send = b_post_send;
}

/* Generated buffer function. */
void create_buffers()
{
    buffer_table = BT_create();
    BT_init(buffer_table);

    buf = create_buffer();
    buf->arrival = (void *)buf_data_arrival;
    buf->pre_remove = (void *)buf_pre_remove;
    buf->post_remove = (void *)buf_post_remove;
    BT_insert(buffer_table, buf);
}