Network Services for Distributed Computing

A thesis submitted for the degree of Doctor of Philosophy

of the Australian National University

Wanlei Zhou

September 1990
STATEMENT

I here by state that this thesis contains only my own original work except where explicit reference has been made to the work of others.

Wanlei Zhou
ACKNOWLEDGEMENTS

First of all, I would like to sincerely thank my supervisor, Dr. Brian Molinari for valuable advising and many discussions. As the Chairman of my Supervisory Panel, he led me into the beautiful field of distributed and parallel computing research, which I feel so bright now. He taught and encouraged me to think and write in English instead of Chinese. After I started to write, he tirelessly read through endless drafts, made them more precise and correct.

I would also like to express my gratitude to Professor Robin Stanton. As one of my supervisors and the Head of the Department, he made valuable comments and guidelines for my research project and provided me with the necessary facilities for the research. I am grateful to Dr. Chris Johnson for being one of my supervisors and I sincerely thank him for his many valuable comments and suggestions on my research. I am also deeply indebted to my advisors Dr. John Smith and Dr. Malcolm Newey who have constantly encouraged and helped me in all stages of my research. I thank Dr. Peiyi Tang and Dr. Qing Li for their comments on my thesis writing.

I would also like to thank Apollo Computer Inc. in MA, USA and The ANU for providing me a great opportunity to work and study in the Research and Development Laboratory of Apollo Computer Inc. in Chelmsford, MA, USA for three months. This made me aware of how the large commercial software be developed. I thank Dr. David Ortmayer and Jim Perry of the Domain Distributed Service Group in Apollo Computer Inc. for being my supervisors during my working trip.

I thank our secretaries and programmers in the department, Bev Johnstone, Jeannie Haxell, David Hawking, Ivan Dean, and Drew Corrigan for their warm smiles and efficient assistance.
I thank my friends in the Department of Computer Science, The ANU: Seppo Keronen, Charles Loboz, James Popple, Jimmy Wang, Dan Zhou, Qing Yang, Gavin Michael, Dharmendra Sharma, and Ely Albacea. To work and talk with these caring friends has always been a graceful experience.

I thank my parents for their love and encouragement. To be a son they can be proud of has always been a driving force in my study towards a Ph.D. degree.

I sincerely thank my wife Ling Wang and my son Lingdi Zhou for being here to company me. Without their love, patience, support, encouragement, smile, and sacrifices, I would not have completed this dissertation.
ABSTRACT

In this dissertation we study distributed program systems that involve the remote procedure call (RPC) primitives. In particular, it concentrates on a group of goals which have been attracting the attention of both researchers and practitioners. These goals include: easing the burden of distributed programming, provision of tools for distributed debugging, provision of tools for execution time estimation and provision of services for managing transaction-oriented distributed operations. The dissertation describes the design and construction of the following network services and analytic tools that help to meet the above goals.

1. **Tools for the support of rapid prototyping.** We address the problem of overcoming the complexity of constructing a distributed program by implementing a set of rapid prototyping tools which can generate RPC program prototypes from user specified description files. We first present a distributed application model. It consists of three parts, namely, the user interface, the distributed frame and the application modules. According to this model, we then provide three kinds of tools to help the generation of each part of a distributed application. These tools are: a distributed frame generator, a user interface generator and an application module generator. Executable versions of the generated programs can be obtained immediately. Interfaces are provided to connect all these generated programs together to form the prototype of the distributed application program. We also provide a case study of using our prototyping tools to develop a moderate size distributed system.

2. **Distributed debugging tool.** Few distributed debuggers provide facilities for RPC-oriented distributed program debugging. We have developed a distributed monitor to help debugging RPC-oriented distributed programs. Our monitor is event-based. We first define primitive events as those things that relate to the RPC calling and process forks. Then we define several operations which are used to construct combined events from primitive events or previously defined combined events. The monitor is constructed to automatically record all primitive events and
user defined combined events of a RPC-oriented distributed program into an appropriate distributed database. The monitor then determines a partial ordering between events in a same process and in different processes, based on the event timestamps and a notion of cause and effect. Finally, the partial ordering is used in replaying the execution of the monitored distributed program.

3. *An analytic tool for execution time estimation.* One of the important performance metrics for a distributed program is the expected execution time. As servers that export remote procedures of a RPC-oriented distributed program are assumed to be dynamically allocated within the whole distributed system, the execution time of a RPC-oriented program varies according to different allocations. So we estimate the execution time of a distributed program by averaging over all possible remote procedure allocations. An analytic tool serving such a purpose has been developed. It includes a nondeterministic algorithm for estimating execution time of general RPC programs, the explicit solution for some special cases, and the estimation methods for lower and upper bounds of execution time of general RPC programs.

4. *RPC transaction manager.* In some distributed applications, one may want to associate transaction semantics with a single RPC call or even a group of RPC calls. Most existing RPC implementations or proposals do not employ transaction concept but rather employ at-most-once semantics. We have designed a RPC transaction manager based on a model which is suitable for our discussion. The model consists of definitions for RPC operations and rollback operations. At first the RPC transaction manager is designed to manage transactions consisting of a single RPC call. Then the transaction manager is expanded to manage transactions consisting of several parallel executed RPC calls. We also provide informal proofs showing that the tool we designed ensures the transaction characteristics.
Table of Contents

Acknowledgements ............................................................................................................ i
Abstract ................................................................................................................................... iii
List of Figures ............................................................................................................................ x
List of Tables ............................................................................................................................... xii
List of Listings and Algorithms ............................................................................................ xiii
Chapter 1. Introduction ........................................................................................................ 1
  1.1. Historical Notes ........................................................................................................ 1
  1.2. Distributed Systems Overview .............................................................................. 4
  1.3. Distributed Computing ............................................................................................ 6
    1.3.1. Client/Server Model and Remote Procedure Calls ............................................ 6
    1.3.2. Network Computing System ............................................................................. 8
    1.3.3. Distributed Application Model ......................................................................... 11
  1.4 Problems in Distributed Computing ....................................................................... 13
    1.4.1. Program Development Problems .................................................................... 13
    1.4.2. Management Problems .................................................................................... 18
    1.4.3. Other Problems ................................................................................................ 19
  1.5. Thesis Outline ............................................................................................................ 20
Chapter 2. Tools for Rapid Prototyping .............................................................................. 23
  2.1. Rapid Prototyping and its Context .......................................................................... 23
    2.1.1. The Life-Cycle Model ....................................................................................... 23
2.1.2. Computer-Aided Rapid Prototyping ............................................................. 25
2.2. Structure of the Prototyping Tools ................................................................. 27
  2.2.1. The Prototyping Model .............................................................................. 27
  2.2.2. Distributed Calendar: The Example .......................................................... 30
  2.2.3. A Concurrency Primitive ........................................................................ 31
2.3. Distributed Frame Generator ......................................................................... 32
  2.3.1. Syntax ........................................................................................................ 32
  2.3.2. Semantics .................................................................................................. 34
  2.3.3. Implementation ......................................................................................... 37
2.4. User Interface Generator ............................................................................... 53
  2.4.1. Introduction ............................................................................................... 53
  2.4.2. Syntax ....................................................................................................... 55
  2.4.3. Semantics ................................................................................................ 56
  2.4.4. Implementation ....................................................................................... 57
2.5. Application Module Generator ...................................................................... 64
  2.5.1. Introduction ............................................................................................... 64
  2.5.2. Syntax ....................................................................................................... 65
  2.5.3. Semantics ................................................................................................ 65
  2.5.4. Implementation ....................................................................................... 68
2.6. Interfacing the Generators ............................................................................ 70
2.7. Prototyping the Distributed Calendar System ............................................. 71
Chapter 3. Debugging RPC Programs ................................................................ 75
  3.1. Introduction ................................................................................................ 75
3.2. Definitions and Assertions ................................................................. 79
  3.2.1. Primitive Event ................................................................. 80
  3.2.2. Combined Event ............................................................. 85
3.3. Monitor Structure ........................................................................ 88
  3.3.1. Overview ................................................................. 88
  3.3.2. Debugging Library ............................................................. 90
  3.3.3. Managing Server ............................................................ 91
  3.3.4. Controller ................................................................. 93
  3.3.5. Event Definition File .......................................................... 95
3.4. Trace Analysis .............................................................................. 98
  3.4.1. Ordering Events .............................................................. 98
  3.4.2. Process Replay ............................................................ 101
3.5. An Example ............................................................................... 102
3.6. Remarks ................................................................................ 106

Chapter 4. Execution Time Evaluation .............................................. 107
  4.1. Existing models and their limitations. .................................... 107
  4.2. The Evaluation Model .......................................................... 109
    4.2.1. Syntactic Issues ......................................................... 109
    4.2.2. Semantic Issues .......................................................... 114
    4.2.3. Performance Measures .............................................. 116
  4.3. Parallel Block Special Case .................................................. 117
    4.3.1. Analysis ................................................................. 117
    4.3.2. Explicit Solution ......................................................... 118
4.3.3. Comparison Between Calculated and Simulated Results ........................................ 120

4.4. Estimation of Lower and Upper Bounds ................................................................. 122
  4.4.1. Lower Bound ........................................................................................................ 122
  4.4.2. Upper Bound ......................................................................................................... 123
  4.4.3. Simulation Results .............................................................................................. 124

4.5. Applications of the Model ......................................................................................... 126
  4.5.1. Example: A Seller-Buyer System .................................................................. 126
  4.5.2. Example: An Extended Seller-Buyer System ............................................ 127

4.6. Remarks ............................................................................................................................ 128

Chapter 5. RPC Transaction Management .................................................................... 130

  5.1. Background and Related Works .............................................................................. 130
  5.2. RPC Transaction Models ............................................................................................ 133
    5.2.1. The RPC Model ................................................................................................... 133
    5.2.2. Single RPC Transaction Model ...................................................................... 140
    5.2.3. Parallel RPC Transaction Model ................................................................... 140
    5.2.4. Accessing Servers ............................................................................................... 141

  5.3. Single RPC Transaction Management ................................................................... 143
    5.3.1. The Manager ......................................................................................................... 143
    5.3.2. Algorithms ............................................................................................................. 145
    5.3.3. Properties ................................................................................................................ 151

  5.4. Parallel RPC Transaction Management ................................................................. 157
    5.4.1. Algorithms ............................................................................................................. 157
    5.4.2. Properties ................................................................................................................ 159
Chapter 6. Conclusions ............................................................................................................. 162

6.1. Summary .......................................................................................................................... 162

6.2. A Computer Science Perspective ................................................................................... 163

6.3. Further Research Directions ...................................................................................... 164

Appendix .......................................................................................................................................... 166

References .................................................................................................................................. 182
List of Figures

1.1. The client/server model 7
1.2. The remote procedure call communication model 8
1.3. Terms used for distributed applications 11
1.4. The distributed application model 12
1.5. Network services for RPC-oriented distributed computing 20
2.1. The computer-aided prototyping model 28
2.2. General prototyping process 29
2.3. Input and output of the distributed frame generator 36
2.4. Processing structure of the distributed frame generator 37
2.5. Input and output of the user interface generator 57
2.6. Menu-tree pointers 58
2.7. The menu screens 63
2.8. Input and output files of the application module generator 67
2.9. Generator connections 70
3.1. Events with the same timestamp value 81
3.2. Events relations 87
3.3. The structure of the distributed monitor 89
3.4. RPC communication 100
3.5. Event graph 106
4.1. The relationship among client program, location broker, and servers 108
4.2. Program flowgraph 112
4.3. Flowgraphs of atoms 113
4.4. The flowgraphs of the three block constructors 113
4.5. Nondeterministism of the algorithm 116
4.6. Flowgraph of the simple parallel block 117
4.7. Average execution time and simulation results 121
4.8. Flowgraph of a complex RPC program 125
4.9. Simulated execution time and lower and upper bounds when \( L \) changes 125
4.10. Flowgraph of the simple example 126
4.11. Structure of the complex example 128
5.1. RPC and its rollback operation 137
5.2. Some possibilities when return is \( US \) 138
5.3. Semantics of the single RPC transaction 140
5.4. Semantics of the parallel RPC transaction 141
5.5. Structure of the RPC Manager 143
5.6. The idea LET table structure 153
A-1. The architecture of the distributed calendar system 169
List of Tables

2.1. Output files for ca.def 40
2.2. Files generated by NIDL compiler 50
2.3. Field type definition 67
3.1. Controller commands 93
3.2. Remote procedure calls of the calendar system 103
3.3. Server events 103
3.4. Client events 104
3.5. Remote predecessor/successor relations 104
3.6. Final predecessor/successor relations 105
4.1. Colours and their usage in Algorithm 4.1 114
List of Listings and Algorithms

Listing 2.1. Server definition file syntax 33
Listing 2.2. Server definition file for calendar server ca.def 41
Listing 2.3. NIDL file for calendar server ca.def 42
Listing 2.4. Server driver file for calendar server ca.def 43
Listing 2.5. Client driver file for calendar server ca.def 46
Listing 2.6. Include file for calendar server ca.def 51
Listing 2.7. Interface definition file syntax 55
Listing 2.8. Menu data structure 58
Listing 2.9. Interface definition file example 59
Listing 2.10. Menu driver file 60
Listing 2.11. Menu include file 62
Listing 2.12. Database definition file syntax 66
Listing 2.13. Database definition file example 69
Listing 3.1. Event data structure 91
Listing 3.2. Event definition file syntax 96
Algorithm 4.1. 115
Algorithm 5.1. 146
Algorithm 5.2. 147
Algorithm 5.3. 148
Algorithm 5.4. 149
Algorithm 5.5. 150
Algorithm 5.6. 151
Algorithm 5.7. 158
Chapter 1
Introduction

1.1. Historical Notes

The strictly sequential computer model was proposed by von Neumann and his contemporaries to avoid a large number of difficult problems intrinsic to parallel computation. In this model, one instruction is processed at a time. The sequential model of computation subsequently dominated programming for many years.

In the 1950's, computer designers began to depart from this strictly sequential model of computing. For example, Input/Output controllers were introduced into computers to work "concurrently" with the central processing unit (CPU). This relieved the fast CPU from waiting on the slower Input/Output devices. In modern multiprogrammed systems many separate jobs can be loaded and run simultaneously, with the system relying on the services of several logically distinct but communicating processes to deal with the stream of jobs. By connecting terminals to host computers, one can access the host computer from distant locations. This can be viewed as the first "geographical" distribution of a computer system.
Then in the early 1970's, a few years after the emergence of minicomputers (which in some instances were designed as single-user computers for software development), the motivation for distributed systems became apparent. By connecting computers with each other, computer networks are created. Since then, especially during the 1980's, we have seen an explosion of the interest in and availability of distributed systems. We mention here three of the most important reasons for this growing interest:

1. Sequential machines have approached the fundamental limits of the computing power they can provide [Bodlaender87]. The cost of VLSI processor and memory components has fallen dramatically and is continuing to do so, and high-speed networking technologies are now widely available at moderate cost. Further contributions to the speeding up of computer systems seem to be possible and economical only if parallelisation of the hardware and of the programs is used. This leads to a distribution of hardware units, control units, and software.

2. Increasingly diverse application facilities are required by users [Coulouris88]. For example, users may want to perform network-based communication, complex information retrieval, and interactive graphics operations. All these operations involve many computing resources and various kinds of information located in different locations. The natural architecture for these applications involves the distribution of facilities to different sites, connected by communication links. Each facility may include computing power, information, programs, Input/Output devices, and other needed resources.

3. Our dependence on computing systems today is so great that computer failures may lead to life-threatening consequences and significant economic impact [Maxion87]. Fault tolerance is now a required attribute of computing systems. One of the possible ways to achieve fault tolerance is to duplicate the hardware components in the system, such as processing units [Zhou90b]. In this way we not only achieve a speed up of the computer system, but we also may obtain a more reliable
computer system: when some components malfunction and others do not, the sys­
tem as a whole may continue to work properly by letting the "good" components 
also do the work of the "bad" ones.

The development of single-user workstations, file servers, and high-speed local 
networks during the 1980’s is an important impetus for distributed system research and 
development. With a single-user workstation, a user can have dedicated processing 
power, enabling an application program to maintain an interactive dialogue with the 
user without interruption. Through the high-speed network, the user can access 
resources in file servers, which manage and store shared information and user files.

The earliest development of a workstation-based distributed system (single-user 
workstations, servers, and high-speed local network) was at the Xerox Palo Alto 
Research Center in the period 1971-1980 [Coulouris88]. The first workstation 
developed was the Alto and came into general use in 1973 [Thacker81]. Then begin­
nning from 1980, early stage commercial workstations, such as the Apollo Domain 
DN100, DN300 and DN600, Sun Microsystems Sun-1 and Sun-2, and Apple Macintosh 
II became available.

Since 1980, there has been a rapid expansion of the research and development of 
distributed systems. For example, the LOCUS [Popek85] system developed at the 
University of California, Los Angeles, the Network File System by Sun Microsystems, 
the Argus [Liskov83] integrated programming language and system, the Cambridge 
Distributed Computing System [Needham82], the Apollo Domain system [Leach83], 
the Amoeba system [Mullender85] at the Vrije (Free) University and CWI, Amsterdam, 
and the Mach distributed system [Mason87] developed at Carnegie-Mellon University. 
Three of these distributed systems, the Sun NFS, the Apollo Domain, and the LOCUS 
have been developed to commercial products.
1.2. Distributed Systems Overview

For clarity we use the following terminology conventions. The term *architecture* will be used to describe hardware aspects while the terms *model*, *structure* and *computing* will be used to describe software aspects. The term *system* will be used to describe the combination of hardware and software.

A common feature of all distributed systems is that there are multiple processes, processors or computers that communicate and cooperate with each other. LeLann [LeLann81] discussed the aims and objectives for distributed systems and notes some defining characteristics. We draw the distinction between a parallel system and a distributed system. In a parallel computer system many identical processors will work on the same problem simultaneously, the processors work synchronously, and the communication delay time is not large compared with the computation time of a processor. The processors are connected in an "interconnection network". In a distributed computer system the processors (or computers) will work asynchronously, the processors may be of different types, and the communication delays may be large in comparison to the computation time of a processor. The processors are connected in a "computer network".

The implementation of distributed systems is constrained by the underlying hardware and basic communication software. In this dissertation we only consider systems that are produced by combining a communication network (such as ethernet [Shoch83] or token ring [Bux83]) with conventional processors (computers). Alternative architectures such as dataflow machines [Sharp85], systolic systems [Kung88] fall outside the scope of the dissertation.

One of the important classes of distributed systems is the *Local Area Network* (LAN). In the usual understanding, the most significant difference between *Wide Area Networks* (WANs) and LANs is the scale. For WANs, the distances between the loca-
Chapter 1. Introduction

tions of the computer systems are large as in, for example, world-wide networks. For
LANs, the scale is much smaller. The smaller distances facilitate different technologies
for faster information transfers between the sites involved. Typically the computer sys-
tems connected in a LAN are workstations and personal computers, within a short dis-
tance of each other. For example, all the nodes of a LAN may be located in the same
building or in several adjacent buildings.

Kirkpatrick [Kirkpatrick89] gives a more precise definition for a LAN. According
to his definition, a LAN is no longer a purely local matter. It can span as large as a geo-
ographical area as a continent, in which there are two or more LANs connected together,
but there is a single address space. Essentially, two characteristics are unique in LANs
and not in WANs. Firstly, all nodes in a LAN may listen to all information, while in a
WAN, not all nodes can listen to the information on the WAN. Secondly, a LAN has a
single address space at Layer 2 (data link layer) of the OSI ISO reference model
[Standard84] whereas a WAN has a single address space at Layer 3 (network layer).

Because of these characteristics, a LAN can send larger packets than a WAN. It
has small latency, lower error rate (lower noise), and higher bandwidth. Usually a LAN
is managed by a single economic entity.

Some authors define the consequences of distribution in terms of separation and
transparency ([Popek85], [Coulouris88]). The separation of components is an inherent
property of distributed systems. Separation allows the truly parallel execution of pro-
grams, the containment of component failures and the recovery from faults without
disruption of the whole system, and allows the incremental growth or contraction of the
system through the addition or subtraction of components. Because of separation, tech-
niques for communication and integration become necessary.

Transparency is defined as the concealment of separation from the user and the
application programmer. With the help of transparency, the programmer can perceive
the system as a whole rather than as a collection of independent components.
Chapter 1. Introduction

Transparency is one of the major influences on the design of distributed system software.

1.3. Distributed Computing

Distributed computing is concerned with the software aspects of distributed systems. In terms of the ISO OSI reference model, it is about the application layer and presentation layer.

Many distributed systems in use and under development are based on workstation/server model [Coulouris88]. In this model, each user is provided a single-user computer, known as a workstation. Application programs are executed in the user's workstation. File servers store and manage shared data and other specialised device servers manage expensive devices such as laser printers, plotters and scanners. One possible extension to this model is to have different types of workstations, or even some multi-user computers connected in the network, resulting in a heterogeneous computer system [Notkin88].

1.3.1. Client/Server Model and Remote Procedure Calls

Most programs executing in the above distributed systems have the client/server structure ([Svobodova85], [Leeuwen89]). The rationale for client/server model was described by Gentleman [Gentleman81]. In this model, server processes manage objects and client processes access these objects by using communication facilities provided by the system and servers. Servers are shared by many client processes. Figure 1.1 pictures this model. Here server $S_1$ and server $S_2$ manage objects $O_1$ and $O_2$, respectively. Client $C_1$ and client $C_2$ access these objects by using the remote operations provided by the servers. These servers and clients may reside on different hosts within the LAN.
Two classes of communication models are frequently used in distributed computing [Gifford87]: the message passing based model and the remote procedure call (RPC) based model. Program parts (clients and servers) can use the operations provided by these models to communicate each other. In the message passing based model, communications are performed by using two primitives: send and receive. One of the major advantages of the message passing based model is that the send operation does not necessarily block the source process waiting a response from the target process. So the interprocess communication in this model can have a lot of forms instead of just request-response pairs.

The RPC based model allows a programmer to call a procedure located at a remote computer in the same manner in which a local procedure is called [Nelson81]. One of the major disadvantages of the RPC model is the limitation of the communication form — only the request-response form is allowed, that is, the calling process is blocked (suspended) until the call completes and reply has been received, as shown by Figure 1.2.
1.3.2. Network Computing System

The RPC based communication model has been investigated by a number of researchers. Nelson’s PhD thesis is still the most complete description of RPC facilities [Nelson81]. Since that time, a number of RPC-based systems have been built. The classic paper of Birrell and Nelson [Birrell84] describes the RPC mechanism they built for the Cedar programming environment, using the datagram communication protocol over the Xerox internet. The Argus language developed at MIT by Liskov and her colleagues [Liskov83] integrated RPC as a part of the language. The Admiral system [Wilbur87], developed at University College, London, by Bacarisse and Wilbur, embeds the RPC construct into the C language. It consists of an interface language and a library of C procedures to be used in client and server programs. One of the most successful
commercial RPC based tools is Apollo’s Network Computing System (NCS) [Apollo87]. It can be used in several programming language environments (e.g., C and Pascal).

NCS is a set of software tools for distributed computing. The foundation for this system is the Network Computing Architecture (NCA) which supports situations where both data and execution are distributed across one or more heterogeneous networks. The following components are provided by the NCS to assist the development and execution of the programs related to the NCS [ZhouRep88]:

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

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

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

† NCS and Network Computing System are trademarks of Apollo Computer Inc.
Chapter 1. Introduction

linked with the server stub and the client program is linked with the client stubs. Now
the server program can run on the remote host and the client program running on the
local host can execute the remote procedures in the same way as it requests local pro-
cedures.

One can notice that we did not mention the location broker above. A small and
specific application needs to have no recourse to the location broker because the client
program knows where the remote services are located. The location broker is very use-
ful in general, however. Usually, a server program must register all of its services with
the location broker. The client program can then find the service through the location
broker. After the client finds the location of the service, it then calls the service directly.
This is called unbound (or allocated) calling. We will use this as the standard calling
semantics. Of course, NCS also supports other calling semantics, such as bound-to-host
and fully bound calls [Apollo87].

There are two kinds of location brokers, called the Global Location Broker (GLB)
and the Local Location Broker (LLB), respectively. The LLB provides the services
information of its local host, and the GLB provides the services information of the
whole network. When the difference between these location brokers is not important,
we will use the term location broker (LB) to specify them. There are some system calls
provided by the NCS to manage the location brokers. Because there may be many ser-
ices registered in a location broker, a unique naming facility called Universal Unique
Identifier (UUID) is employed. Each service has to be assigned a UUID before it is
registered to the LB. These UUIDs are used by NCS to distinguish one service from
another. The NCS provides system calls to support this function.

A distributed program in NCS can be functionally divided into two parts: the
server part and the client part. Each part can be located on any host in the network.
Usually, a server part manages an object, and a client part accesses the object by using
the remote procedures provided by the server. A RPC-oriented program (in short, a
Chapter 1. Introduction

Remote Procedure Call (RPC) program) may consist of several servers and clients, and all these parts of the program work together concurrently on the programmer's task. A server or client can fork to several processes if necessary.

1.3.3. Distributed Application Model

Figure 1.3 indicates the terms we have used to describe our distributed applications. There are two classes of distributed applications (or equally, distributed programs): message passing-oriented distributed applications and RPC-oriented distributed applications. As we have seen, a RPC-oriented distributed application can be again divided into server program parts and client program parts. We shall simply call them as program parts (PP). From now on, when we mention distributed application, we mean RPC-oriented distributed applications unless we state it explicitly.

Distributed Applications (Programs)

RPC-Oriented Distributed Applications

Message Passing-Oriented Distributed Applications

Server Programs

Client Programs

Figure 1.3. Terms used for distributed applications

One important class of distributed applications is distributed information system (IS) applications. In a distributed IS application, there is usually a number of computers and processes managing some shared information, such as databases. User programs
access these computers and processes to obtain the information the user needs, or to update the stored information through these computers and processes. Time in a distributed IS application is not as strict a requirement as in distributed real-time applications. This dissertation is interested in distributed IS applications. From now on, when we mention distributed application, we mean distributed IS applications. Figure 1.4 is a generic model of distributed applications.

Figure 1.4. The distributed application model

According to Figure 1.4, a distributed application consists of several client programs and several server programs. Usually a server program is located on a remote
Chapter 1. Introduction

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

— **User interface.** This deals with the interactions between the client program and the user.

— **Distributed frame.** This performs the communications among all the co-operative parts over the LAN.

— **Application modules.** They manage the objects and perform operations.

### 1.4. Problems in Distributed Computing

To use distributed systems correctly and efficiently, many problems must be dealt with. Some of the problems addressed in the dissertation are described below.

#### 1.4.1. Program Development Problems

An essential problem in distributed computing is the design of algorithms and the implementation of corresponding programs. The equivalent problem for sequential program development is nowadays reasonably managed and a large number of techniques exist. Due to the fact that many parts of a distributed program will work concurrently, interact with each other, and influence each other in a sometimes
nondeterministic way, distributed programs seem more difficult to understand than sequential programs. Consequently, distributed programs are more difficult to design and program. We mention the following interesting aspects.

**Distributed Programming**

Programming a distributed program is much more difficult than programming a sequential program because the former involves communication and cooperation among processes. For instance, in programming RPC-based programs, one has to write server and client communication stubs, modules of RPC procedures, server programs, client programs, user interfaces, and application modules. Although some facilities are provided to ease the burden of distributed programming (for example, NCS provides a NIDL compiler (see Section 1.3.2) which can produce server and client communication stubs from a specification file), the programming process for distributed programs is still more complex than for sequential programs.

Suppose we are going to build a calendar database management system [ZhouRep89] (see Section 2.2.2 and Appendix A for details) by using NCS. We may use the following steps to implement the system:

- Define the program structure and remote procedure interfaces. Usually we use a server program to manage the calendar database, and allow several client programs to access the database through the remote procedures exported by the server. Then, the server program must have the ability to maintain the database operations, and the client program must have the ability to interact with users. After the definition of the program structure, the interfaces for all remote procedures that the server is going to export can be defined. This is actually the interface between the server and the client programs based on the programmer’s point of view.
Chapter 1. Introduction

- Generating stub modules. The definition file is written in the NIDL language. After using NIDL compiler, the server and client stub modules will be generated. Now, the client program can view the client stub module as the "local version" of the server program, and the server program can view the server stub module as the "local version" of the client program. These stub modules perform most of the lower level communications between server and client.

- Write server program modules which perform the real work of the remote procedures. For example, if a remote procedure corresponds to "insert a new record into the database," the real procedure which does the insertion of the record must be built as a part of the server program.

- Write client program modules which make use of the remote procedures and interact with users.

- Combine server program modules together to form the server program.

- Combine client program modules together to form the client program.

The RPC program is then ready to execute. Several problems may arise during the development process.

1. It takes a long time before an executable version of the system is ready for testing because the programmer has to write all those modules before they can be combined and tested. During this time, the user requirement may change. If the modules are written by several programmers simultaneously, the interfaces between them are an intrinsic problem for group development.

2. Because a distributed program is relatively complex in the sense that cooperating modules and programs are involved, it is very difficult to manage the correctness of all these parts. In the above we mentioned the process involved in the creation of one server and its client programs. If there are several servers in the system (which is usually the case), the programmer has to do the same thing for each.
server and its client programs. If a client program is going to use several servers’ remote procedures simultaneously (which is also typical in distributed programming), the situation becomes even worse. The memory of human beings is limited. When there are too many things to be remembered during the development process, it is very easy for programmers to make mistakes. When there are errors in the program, it is very difficult to find them because of the complexity of the system.

3. It is very difficult to accommodate module changes during the development process. For example, if one of the remote procedure interfaces is changed, the programmer has to change all the program modules involved such as the server program modules which export the remote procedure and the client modules which use the remote procedure. Of course module changes are intrinsic to program development processes.

Our solution to these problems involves a set of rapid prototyping tools for distributed program development. By using these tools, one can quickly build an executable version of the system. The program modules generated by these tools are relatively correct because these tools are tested. Also these tools allow changes in interface and modules to be accommodated. For example, if a remote procedure interface is changed, the prototyping tools will automatically change all the appropriate modules in the new prototype.

Distributed Program Debugging

Debugging is a process of isolating, diagnosing and correcting program errors [Fairley85]. For large systems this is one of the most costly phases in the program development process [McDowell89]. Debugging a distributed program is usually much more difficult than debugging a sequential program. The concurrency and communication among the concurrent parts make debugging of distributed programs difficult
Chapter 1. Introduction

[Miller86]. We list the following typical difficulties:

- Because several parts (programs or processes) may execute on different hosts simultaneously, some events will occur at the same time. But in general there is no global timing base that can be provided for all the hosts of a distributed system, so we cannot completely order the events in a distributed program nor accurately measure when these events occur. This is in direct contrast to the situation with the debugging of a sequential program.

- Because of the finite but nondeterministic time needed for communication between hosts, there is no way to obtain a snapshot of a distributed program’s global state. Nor is it possible to have instantaneous change of control for all parts of a computation on different hosts. That means that methods such as breakpoint and stepping are not as effective as in sequential debugging.

- Because of the processes in a distributed program work concurrently, and the time needed for their activities is nondeterministic, it is almost impossible to re-do an execution to obtain the same behaviour. This breaks one of the fundamental assumptions of the sequential debugging [Garcia85].

- Because of the communication time between hosts is much longer than the computation time within a host, an error condition in a host may affect many processes before it is detected or noticed by the programmer. This complicates the determination of where the error really occurred. Although this error propagation effect also exists in sequential debugging, the affected area can be much smaller than in distributed debugging because of the shorter latency time.

It is evident that the ability to trace communication events and concurrent events (as well as other events) that occur during a distributed program’s execution is fundamental to any debugging tool. Most existing studies of distributed debugging and monitoring are based on the message passing model. As the RPC model become more and more popular, the requirements of RPC-oriented program debugging are increasing.
Chapter 1. Introduction

Our solution to this is a RPC-oriented debugger which records all interesting events of a distributed program into an appropriate database. A partial ordering among events is then built. This ordering relation can be used to trace and replay the program’s execution. The debugger and its database is distributed. It can monitor several distributed programs simultaneously.

Performance Evaluation Problems

We have mentioned the construction and debugging of distributed programs. After the distributed program is built, we may want to know how well it works. This involves the performance evaluation of distributed programs.

One of the important performance metrics for a distributed program is the expected execution time. Most existing evaluation methods use queueing theory to analyse the expected execution time. But this time corresponds to the system’s viewpoint and averages over all possible jobs. In the other hand, a user is often interested in the execution time of his own job, which is a quite different viewpoint [Qin89].

We have developed a model for evaluating the execution time of a particular RPC program. For simple programs we can derive explicit solutions. For complex programs, a nondeterministic algorithm as well as the lower and upper bounds of the execution time are developed.

1.4.2. Management Problems

A distributed application program consists of many concurrent processes, and the purpose of the program is to have all these concurrent parts work together to achieve a common goal. Therefore the management of the cooperation among these concurrent
parts are critical. We mention some of the interesting problems which arise with the management of a distributed program.

1. **Resource sharing.** Processes and program parts may want to use the same resources at the same moment. Therefore facilities must be dedicated to the fair and efficient allocation or occupancy of the resources, for instance, to avoid deadlock and starvation, and to maintain mutual exclusion.

2. **Fault tolerant computing.** For many problems in distributed computing one may want to have solutions that still behave correctly if one or a small number of processors/links/processes fail.

3. **Atomic actions.** In some distributed applications, one may want to keep a group of operations atomic. That is, the effect of these operations is all-or-nothing. Because the involved operations may spread to the whole LAN, this task is difficult.

4. **Accommodation of heterogeneity.** In a LAN there can be several types of computers. How these computers talk to each other is very important. But unfortunately, almost all existing RPC tools provide a closed environment. For example, a NCS/RPC can only talk to NCS/RPCs [Notkin88]. This brings many difficulties to the programming of heterogeneous computer systems. So interfaces between different RPC tools are needed [Zhou90a].

1.4.3. **Other Problems**

The above list is only a subset of the problems in distributed computing, and other important problems exist. These include, load balance within a LAN, "global" state and ordering, communication protocols, the mapping of problems onto a known distributed architecture, and the verification of distributed programs.
1.5. Thesis Outline

In this thesis we address several of the problems in distributed computing that have just been outlined. In particular we design and implement a set of RPC-oriented network services which will help to solve some of the above problems. The main goals of this effort are:

1. Easing the burden of distributed programming.
2. Provision of tools for distributed debugging.
4. Provision of services for managing RPC transactions.

Figure 1.5 indicates the relationship of the above services and underlying facilities. All work is based on the Apollo NCS.

![Diagram](image-url)
Chapter 1. Introduction

The contents of the following chapters are:

- Chapter 2 describes a set of prototyping tools which can generate RPC program prototypes from user specified description files. This approach frees programmers from many of the details of programming, such as the establishment of communication during system initialisation and implementation of the user interface ([Zhou90c], [Zhou90d], [Zhou90f]). At first existing software development methods are described and the prototyping model for distributed information system applications is presented. The design and implementation of the prototyping system are then described together with application examples from the distributed calendar system.

- Chapter 3 presents a distributed debugger. It can debug and monitor a distributed program system ([Zhou90d], [Zhou90g], [Zhou90h]). It first reviews the related works and then gives several definitions that are basic for later discussion. The structure of the debugger is then described and the method of trace analysis is presented.

- Chapter 4 describes a technique for the estimation of the execution time of RPC-oriented programs ([Zhou89], [Zhou90e]). After introducing the RPC program model, a nondeterministic algorithm for calculating the execution time of a general RPC program is presented. Although it is impossible to obtain the explicit solution for a general RPC program at this moment, an explicit solution for some special cases is presented. Then, lower and upper bounds for the general model are derived. Applications of the theory are also presented.

- Chapter 5 gives the description of a RPC transaction manager which ensures the transaction characteristic for a group of RPC operations [ZhouRep90b]. After a survey on the existing work, we present a RPC model which is suitable for our discussion. Based on this model, a RPC manager is designed to manage a transaction consisting of a single RPC call. Several properties of the manager are then argued.
Chapter 1. Introduction

By extending the RPC model, a transaction manager is designed for managing several RPC calls executed in parallel. Properties of the transaction manager are also described.

Finally, Chapter 6 provides a summary of the thesis.

The best way to describe the application of the environment is through examples. I use a moderate size distributed application as the example. It is a distributed calendar system and was developed during a working trip to Apollo Computer Inc. in MA, USA. To avoid describing its structure at several places in the dissertation, a description of the system is provided as an appendix.
Chapter 2
Tools for Rapid Prototyping

2.1. Rapid Prototyping and its Context

2.1.1. The Life-Cycle Model

For the past twenty years or so, software system development has been based on
the software life-cycle model (or, waterfall model) [Boehm76]. This model essentially
advocates that software projects should consist of a number of distinct phases [Boar84].
These are: specification, design, implementation, testing, operation and maintenance.
This model has been modified by a lot of researchers since its inception. But the central
idea remains unchanged, that is, all variations keep the linear structure, and each phase
begins only when the previous phase has been completed.

Several assumptions underly the life-cycle model ([Agresti86], [Boar84]):

(1). Computers are an expensive resource, and their access should be preceded by
careful planning so that the time on computers would be used effectively.

(2). A complete, concise and consistent specification of a proposed system can be pro-
(3). Successful software was developed by successively achieving subgoals at each phase of the waterfall.

The life-cycle model works very well when the application is both well understood and supported by previous experience, but in general, it has many deficiencies which are too serious to be ignored [Hekmatpour88]. The reason is that some assumptions made by life-cycle model are no longer true. We list the following deficiencies.

(1). The computers are much cheaper than 20 years ago and there are many programming tools available today. The access to computers and software are no longer limited to people with specialised skills. The relative cost effectiveness of computer hardware technology has increased by a factor of 10,000 since 1970 [Musa83]. A wide array of software development tools are being used for every phase in the life-cycle model. Of greater significance, some current tools and environments logically span several conventional phases, thereby challenging the usefulness of the life-cycle's partitioning into phases [Agresti86].

(2). It is impossible to rigorously specify all requirements of a distributed IS application. Many authors have pointed out that a complete, concise and consistent specification of a proposed system is impossible ([Swarthout82], [McCranken81], [Shaw85], [Parnas86], [Agresti86], [Hekmatpour88]). The main reason is that, as human beings, people need to see examples and have practical experience before they are able to make judgements about the suitability of a proposed system and to recommend revisions. Even if a fine and valiant effort is made of specification, the initial contact with the solution changes the individual's perception of what they want.

(3). It is impossible to clearly divide and therefore achieve each subgoal of the life-cycle model. For example, one cannot state a problem without some rudimentary notion of what the solution should be [Agresti86]. That is, to separate the "what"
Chapter 2. Tools for Rapid Prototyping

of specification from the "how" of design is almost impossible [Swartout82]. Actually, every specification is an implementation of some other higher level specification. Thus simply by shifting our focus to an earlier portion of the development, part of the specification becomes part of the implementation. The other thing is that requirements are often a fuzzy thing in the user's mind. So, the design and implementation according to these fuzzy requirements are often unworkable and need revision. In this case, when applying the life-cycle model to the development of distributed IS application, all phases of the life-cycle often interact instead of falling into a linear structure, and hence the development strategy fails.

(4). In the life-cycle model, the user may have to wait a long time before actually having a system available to him or her, because of the successive achievement of subgoals. But during this time, the user requirements as well as the user environment may change considerably. This may cause frustration. Blum [Blum82] described this graphically: "Development is like talking to a distant star; by the time you receive the answer, you may have forgotten the question."

The importance of these observations is that the life-cycle model reflects the time period in which it evolved. Dramatic changes since then in the environment of the software process are promoting a reassessment of the model, and new development models are needed to fit the evolved technology and changed application domains.

2.1.2. Computer-Aided Rapid Prototyping

Computer-aided rapid prototyping has been suggested as an alternative scheme to overcome the deficiencies of the life-cycle model for the development of IS applications ([Blum82], [Boar84], [Boehm84], [Luqi88], [Luqi89], [Balzer89]). In this approach, a range of computer tools is used to help the developers to generate prototype
programs. When the proposed distributed IS application is too complex; when the user interface is an important part of the system; when the requirements cannot be completely specified at the beginning; or when there are too many uncertainties about the proposed system, rapid prototyping is a suitable model for development.

At the least, rapid prototyping is suitable in the following aspects.

- To clean up ambiguities in the specification of a distributed IS application. When developers and users look into the system prototype together, they can find where the misunderstandings are between them and can address them.

- To make the user interface more friendly to users. When creating a user interface, developers usually impose their own judgement. That is, many features of the system are in the developer's mind-set and they may not specify or explain those features in the user interface. An expert cannot think as a computer layman. So the user interface created by a computer expert is usually not suitable to users. By using the system prototype, both the expert and the user can discuss this together.

- To help users specify their wishes. Simply working through specification documents, users may not be able to specify their wishes. By working with the system prototype, they may be able to do so.

- To cope with changes during the development of the system. Because the prototypes are generated by tools, it is easy to cope with design changes.

Several approaches to rapid prototyping exist [Hekmatpour88]. In throw-it-away prototyping, the prototype system is used for a limited period, and is usually used for requirements analysis and specification. After that, it is thrown away. The rapid development of the prototype is the greatest need, while the efficiency of the prototype is of little importance. A second approach is called evolutionary prototyping. Here a system grows and evolves gradually. At first, only those parts of the system that are well understood are developed, and the prototype then evolves as the understanding of the whole system becomes clearer during the use of the first prototype. This is very
suitable for gradually introducing a new system into an organisation and for coping with the changes that take place within the organisation as a result of using the system. This method is the most attractive model for IS applications. It is the prototyping method used in this chapter.

In all cases, a prototype must be a "working model" of the proposed IS application. The challenge is then to develop the related software tools that will support the prototyping process. In the next section, we present several tools that help the development of distributed IS application prototypes.

2.2. Structure of the Prototyping Tools

2.2.1. The Prototyping Model

Based on the distributed application model (described in Section 1.3.3 and schematically indicated in Figure 1.4), we indicate in Figure 2.1 a strategy for the development of distributed IS applications. According to this model, the prototyping process for a distributed application can be divided into three related activities:

- **Distributed frame prototyping**, 
- **User interface prototyping**, and 
- **Application modules prototyping**.

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

In this scheme a distinct tool is provided for each prototyping activity. For example, when prototyping the distributed frame (see Section 1.3.3 for the description of distributed frames), a *distributed frame generator* is used to generate the program modules...
and to test drive programs; when prototyping the user interface, a user interface generator is used to generate the interface program from a straightforward definition file; the application generator is used to generate application modules. Now database management operations are only provided in the application module generation. All these prototypes are relatively independent and can be tested separately with the participation of users. This gives the users a good chance to remove ambiguities in system requirements and to rethink and reorder their needs. After each part of the prototype is tested, their combination forms a working prototype of the whole program and can be tested again by users and developers. Further changes can be made easily because the computer-aided generators are available.
Knowledge-based techniques [Mitchell84] are used in implementing the distributed frame generator. It uses queries as well as server definition files to obtain information of the distributed frame from the user. The knowledge and transform mechanisms of the generator are implemented by using CLIPS ([NASA88], [Giarratano88]), and the C language is used to help the implementation. The other two generators are implemented simply by using the `yacc` utility and the C language.

![Figure 2.2. General prototyping process](image)

The general prototype generating process is indicated in Figure 2.2. At first the programmer creates one or more definition files according to the application. Then these files are sent to the appropriate generator where the test programs are generated. After that, the programmer can execute these test programs and make changes to the definition files on the basis of the results of the execution. Then, the changed definition files are sent to the generator again. When the programmer judges that the programs are correct, the final generation phase is chosen and the appropriate prototype modules are generated. By combining the distributed frame prototype, the user interface prototype and the application module prototypes, the final distributed program is built up.
Chapter 2. Tools for Rapid Prototyping

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

2.2.2. Distributed Calendar: The Example

As mentioned in Section 1.5, we use a distributed calendar system as a non-trivial example. The system has three kinds of databases [ZhouRep89]:

- A name database. This contains the names and corresponding UUIDs (see Section 1.3.2 for an explanation of UUID) of all legal users of the system.
- A meeting-room database. If a meeting room is booked for a meeting, there will be an entry in this database which contains the room name, the meeting time period, and other information.
- A set of calendar database files. There is one calendar database file for each group of users (e.g. all users within a department). If a user has a meeting, there will be an entry in his calendar database file which contains the user name, the meeting time interval, a lot of participants, and other information.

For each database, there is one server which maintains it. We call these a register server, a room server, and a calendar server, respectively. All these servers run "forever" in the network. Figure A-1 in appendix A describes the structure and usage of the system.
When a user logs into the system, his user name is used as a key to check his calendar database location (UUID) from the name database. The relevant calendar database is then searched to see if there are any existing meetings for the user. When a user issues a meeting request, the system first checks the room database to see if the room is available for the nominated period. If it is free, then it determines the calendar database UUIDs of all the participants from the name database and checks if they are all free during the meeting period. If the above check passes, an appropriate entry will be stored into each participant’s calendar database.

2.2.3. A Concurrency Primitive

One of the disadvantages of the RPC based communication model is that it blocks the calling process during a RPC call until the call returns. In many cases, however, it is desirable to have several RPC calls executed concurrently. For example, when we want to engage ten participants in a meeting, we would like to have a facility which can make these ten RPC calls at the same time instead of in sequence. We introduce a concurrency primitive called COBEGIN-COEND into the NCS. This primitive has the following format:

```
COBEGIN(n);
P_1;
P_2;
...
P_n;
COEND(n);
```

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

COBEGIN(n);
    for (i=1; i<=n; i++) p_i;
COEND(n);

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

When a user accesses several RPCs concurrently, he may want these RPCs be executed as an atomic action. That is, either all of these RPCs are executed successfully or none of them is executed. The latter case corresponds to an error condition in one or more of the calls. The other successful RPCs in this case are rolled back. We will discuss this in Chapter 5. In this chapter we will not consider the atomicity of these concurrent RPCs.

2.3. Distributed Frame Generator

2.3.1. Syntax

The purpose of the distributed frame generator is to generate distributed frames for server and client programs according to server definition files (SDF). We give the syntax of a server definition file in Listing 2.1.

We use a modified BNF [Alagar89] to denote the syntax of this and other definition files. In this notation, non-terminals are denoted in ordinary font and terminals are denoted as in 'bold' font, while the symbol ::= denotes defined as. Three operators are involved, namely:

- the construct \{ x \} means that \( x \) is replaced an arbitrary number of times,
- the construct \[ x \] means that \( x \) is optional, and
Chapter 2. Tools for Rapid Prototyping

— the construct $x | y | z$ means that one of the items is selected.

Listing 2.1. Server definition file syntax

```
SDF ::= BEGIN
  [ HEADER ]
  [ CONSTS ]
  [ STRUCTS ]
  FUNCS
END

HEADER ::= Server Name: variable ;
  [ ATTR ; ]
  Register string: string ;
  Communication Protocol: variable ;
  Maximum Queue: integer ;

ATTR
versionNo ::= Interface Attr: uuid, versionNo

CONSTS ::= CONST { CONST }
CONST ::= variable = integer ;

STRUCTS ::= STRUCT { STRUCT }
STRUCT ::= structure-declarator

FUNCS ::= RPC Functions: RPCS
RPCS ::= RPC { RPC }
RPC ::= Name: string ; PARAMS
PARAMS ::= { PARAM }
PARAM ::= Param: CLASS: declarator ;
CLASS ::= in | out | in_out
```

The following notes apply to Listing 2.1.

1. The non-terminals variable, integer, and string have the same meanings as in the C programming language.

2. The non-terminal declarator has the same meaning as in the C programming language.

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

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

2.3.2. Semantics

A server definition file is defined as an optional HEADER part followed by FUNCS part, with constants and data structure definitions also optional. The HEADER includes the server’s name, an optional interface attribute, a register string, a communication protocol, and a specification of maximum queue length. The server’s name is defined as a variable in the C language. This name will be used in the NIDL file (Section 1.3.2) as the interface name.

The interface attribute, if chosen, is a uuid followed by a version number. Please refer to Section 1.3.2 for the meaning of a uuid. If the ATTR part is empty in the server definition file, the prototyping tool will generate an appropriate uuid and set the version number to 0, otherwise the user-provided uuid and version number are used. The version number is used by the NCS to distinguish between various versions of the same server. Only clients with the same version number as the server can access the server’s remote procedures, otherwise an exception will be raised in the client and no remote procedure will be executed in the server.

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

The communication protocol part is used to define which communication protocol is to be used. The NCS currently provides two kinds of protocols, namely, the DDS and IP [Apollo87].
Chapter 2. Tools for Rapid Prototyping

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

The CONSTS part defines the constants used in the interface definition and is self-explanatory.

The STRUCTS part defines the data structures used in the interface definition. It is almost the same as defined in the C language, except that only simple types, such as integer, character, string and double, are allowed inside the structure.

The FUNCS part defines the remote procedures of the server. At least one remote procedure must be defined. Each remote procedure is defined as a name part and a parameter (PARAMS) part. The name of a remote procedure is simply a variable. There can be several parameters, each consisting of a class and a declaration. The class can be in, out, or in_out, which tells the NCS system that the parameter is used for input, output, or both, respectively. The declaration part is the same as in the C language.

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

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

The input to the generator is several server definition files, one for each server. We say the generator is in single-server mode if there is only one SDF input to the generator. If there are two or more SDFs input to the generator at the same time, then we say the generator is in multi-server mode. The output client driver can call any remote procedures of input servers sequentially or concurrently. Figure 2.3 indicates the input and output of the distributed frame generator (the makefile is not shown).

![Figure 2.3. Input and output of the distributed frame generator](image)

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

2.3.3. Implementation

Without loss of generality, we will describe implementation details for the single-server mode. After a programmer sends the server definition file to the generator, the generator first does syntax checking. If no errors are found, several program source files and a makefile is generated. The subsequent processing are specified by the makefile. That is, when using the make utility, at first four new files will be generated by NIDL compiler. They are the server stub, the client stub, the client switch (also used for communication), and an include file for both server and client drivers. After that the executable files of the server and client will be generated. Figure 2.4 indicates the structure of the processing.

![Figure 2.4. Processing structure of the distributed frame generator](image)

The server driver does several jobs after the initialisation. It first obtains a socket address from the operating system and registers itself with the location broker. It then
Chapter 2. Tools for Rapid Prototyping

does some jobs associated with exception handling. After that, it listens to its socket and responds to client calls. Some jobs are also performed before the server is shut down. We outline the action of server driver as follows:

(1). Initialisation.

(2). Set communication protocol to that specified by the programmer, and obtain a socket from the system.

(3). Register the server object with the location broker.

(4). Set the exception handling subsystem.

(5). Listen to the socket and respond to the client requests when necessary (sending the requests to the appropriate RPC procedures and obtaining the returns).

(6). Perform necessary jobs before server shutdown, such as return the socket to the system and unregister from the location broker.

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

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

(1). Obtain the socket address of the server interface by interrogating the location broker.

(2). Allocate (bind) a RPC handle according to the above address.
(3). Allocate an exception handle and prepare the exception handling segment.

(4). Actually call the remote procedure (as for a local procedure call).

(5). Clean up the exception handler and display the returned information.

Of course if the server location is known the calling process will be correspondingly simpler. But we have mentioned earlier in section 1.3.2 that a client program usually knows nothing about the location of a server it wants to call. So, the above steps are necessary in general.

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

An Example

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

What is the program convenience consideration:
(best/good/moderate/low/do-not-care) **best**

What is the program speed consideration:
(highest/high/moderate/low/do-not-care) **high**

What is the program memory consideration:
(smallest/small/moderate/large/do-not-care) **moderate**

What is the program complexity consideration:
(smallest/small/moderate/large/do-not-care) **moderate**

How many different servers? 1
Chapter 2. Tools for Rapid Prototyping

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

Name of the server definition file: ca.def

The first four queries (in the first paragraph) are used for selecting different program modules and implementation methods which may affect the program’s performance. The second paragraph is used to obtain some general information about the server(s). And the last paragraph indicates the input server definition file(s). The server definition file ca.def is given in Listing 2.2.

As we are using the single-server mode, Table 2.1 indicates the generated files.

Table 2.1. Output files for ca.def

<table>
<thead>
<tr>
<th>File Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca.idl</td>
<td>NIDL file of the server</td>
</tr>
<tr>
<td>ca_ser.c</td>
<td>server drive program</td>
</tr>
<tr>
<td>ca_rprocfuncs.c</td>
<td>RPC template file</td>
</tr>
<tr>
<td>dp.c</td>
<td>client drive program</td>
</tr>
<tr>
<td>ca_include.h</td>
<td>include file for server and its client</td>
</tr>
<tr>
<td>make_dp</td>
<td>make file for testing programs</td>
</tr>
</tbody>
</table>

The prototyping tool generates a UUID for the server and sets the version number to 0. The port number will be decided by the NCS dynamically when the server is registered with NCS. Also we added a remote procedure ca$shutdown at the end of the generated NIDL file. This will allow the client to shutdown the server remotely. The generated NIDL file ca.idl is indicated in Listing 2.3.
Listing 2.2. Server definition file for calendar server ca.def

BEGIN

MAXNAMELEN = 20;
MAXVALLLEN = 13;
MAXCMTLEN = 255;
MAXRESULTS = 20;
typedef struct {
    char username[MAXNAMELEN]; /* user name */
    char start[MAXVALLLEN]; /* meeting start time */
    char end[MAXVALLLEN]; /* meeting end time */
    char caller[MAXNAMELEN]; /* meeting caller */
    char comment[MAXCMTLEN]; /* memo of the meeting */
} DISCAL_REC;

RPC Functions:
Name: find_a_cal;
/* find a meeting entry by name & start time */
Param: in: char name[MAXNAMELEN];
Param: in: char start[MAXVALLLEN];
Param: out: DISCAL_REC rec; /* returned entry */

Name: find_all_cal;
/* find all meeting entries for "name" */
Param: in: char name[MAXNAMELEN];
Param: out: DISCAL_REC recs[MAXRESULTS];
/* returned entries */
Param: out: int ret_num;
/* returned number of entries */

Name: add_a_cal;
/* add a meeting entry into the database */
Param: in: DISCAL_REC rec;

Name: del_a_cal;
/* delete an entry from the database */
Param: in: char name[MAXNAMELEN];
Param: in: char start[MAXVALLLEN];

Name: change_a_cal;
/* change the contents of an entry */
Param: in: char name[MAXNAMELEN];
Param: in: char start[MAXVALLLEN];
Param: in: DISCAL_REC rec;
/* changed entry */

END
Listing 2.3. NIDL file for calendar server ca.def

/* NIDL file of server "ca" */

%c
[uuid(495795f1f20c.02.82.38.04.3b.00.00.00), version(0)]

interface ca

{

import 'nbase.idl';

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

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

cas$find_a_cal(
    handle_t [in] h,
    char [in] name[MAXNAMELEN],
    char [in] start[MAXVALLEN],
    DISCAL_REC [out] rec
);

cas$find_all_cal(
    handle_t [in] h,
    char [in] name[MAXNAMELEN],
    DISCAL_REC [out] recs[MAXRESULTS],
    int [out] ret_num
);

cas$add_a_cal(
    handle_t [in] h,
    DISCAL_REC [in] rec
);

cas$del_a_cal(
    handle_t [in] h,
    char [in] name[MAXNAMELEN],
    char [in] start[MAXVALLEN]
);

cas$change_a_cal(
The generated server driver at first registers itself with the location broker, prints out the obtained socket information if the registration is successful, and then sets the error handling segment. After that, it listens to its socket port and serves its clients if necessary. That is, the server program will loop forever at the statement

\[
\text{rpc\_listen(MaxCalls, &st);} 
\]

If any client calls come in, the server will serve them and then return to the loop. The maximum number of concurrent calls to the server is defined in MaxCalls, which in turn comes from the query process or server definition file. If the client call is the remote shutdown call, the server exits from the \text{rpc\_listen} function and unregisters itself from the location broker. After that, the server program exits. The generated server driver is in Listing 2.4:

---

Listing 2.4. Server driver file for calendar server ca.def

/* Server program file of server "ca" */
#include "ca_include.h"

/*
   Several utility functions are omitted here.
*/

static lb_entry_t lbentry;
int SocketAndRegister()
{

handle_t h,
char name[MAXNAMELEN],
char start[MAXVALLEN],
DISCAL_REC [in] rec
);
ca$shutdown(
    handle_t h
);
}

---

Chapter 2. Tools for Rapid Prototyping
Chapter 2. Tools for Rapid Prototyping

status_t st;
socket_addr_t loc;
char name[256];
unsigned long namelen = sizeof(name);
unsigned long port;
extern rpc_epv_t ca$server_epv;

/* Some NCS function calls that allow remote shutdown server, set protocol, and obtain socket and port (assigned to "name" and "port") are omitted here. */

printf("Got socket: %s[%lu]\n", name, port);

/* register to the location broker */
rpc_register(&ca$if_spec, ca$server_epv, &st);
if (st.all != 0) {
    printf("Cannot register to RPC library - %s\n", error_text(st));
    return -1;
}

lb_register(&uuid_nil, &uuid_nil, &ca$if_spec.id, OL,
    RegName, &Loc.sa, Loc.len, &lbentry, &st);
if (st.all != 0)
    printf("Cannot register to GLB - %s\n", error_text(st));

return 0;

int UnregisterBroker()
{
    status_t st;

    lb_unregister(&lbentry, &st);
    if (st.all != 0) {
        printf("Cannot unregister from GLB - %s\n", error_text(st));
        return -1;
    }

    rpc_unregister(&ca$if_spec, &st);
    if (st.all != 0) {
        printf("Cannot unregister from RPC library - %s\n", error_text(st));
        return -1;
    }

    return 0;
}

main(argc, argv)
int argc;
char *argv[];
{
    status_t fst;
    pfm_cleanup_rec crec;
Chapter 2. Tools for Rapid Prototyping

/* socket preparation - get socket, register ... */
if (SocketAndRegister() != 0) {
    printf("Error in socket preparation.\n");
    exit(1);
}

/* report the RPC errors if encountered */
fst = pfm_$cleanup(crec);
if (fst.all != pfm_$cleanup_set) {
    if (fst.all != status_$ok)
        fprintf(stderr, "*** Exception raised - %s\n", error_text(st));
    pfm_$signal(fst);
}

/* Listen to the clients calls */
printf("Listening ... (server -ca-)\n");
rpc_$listen(MaxCalls, &st);

printf("Shutdown. (server -ca-)\n");
UnregisterBroker();
pgm_$exit();

#include "ca_rpcfuncs.c"

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

/* Rpc "find_a_cal" call function of "ca" */
ca$find_a_cal(h, name, start, rec)
handle_t h;
char name[MAXNAMELEN];
char start[MAXVallen];
DISCAL_REC rec;
{
    printf("This is RPC function ca$find_a_cal.\n");

    /* The contents of the RPC function */
}
Chapter 2. Tools for Rapid Prototyping

The generated client driver enters a dialogue with the user. It first requests the number of RPCs to be tested in the current execution. It then asks the input of the RPC numbers for each server, as well as the input arguments of each RPC call. After the verification of this input information and the setting of exception handling segment, the driver calls a RPC parallel execution function `parRun`. If there is only one RPC call to be executed, the single RPC call execution function is called by the `parRun` function. Otherwise, the `COBEGIN-COEND` primitive is used to execute the RPC calls. Listing 2.5 is the client driver.

Listing 2.5. Client driver file for calendar server ca.def

```c
/* Client program file of parallel testing */

#include "ca_include.h"
#include "mhead.h"

#define NumOfServers 1
int NumOfEach[NumOfServers] = {
  6,
};

#define NumOfRpcs 6
char *ServerNames[NumOfServers] = {
  "ca",
};

typedef struct parrpc {
  int sernum; /* server number */
  int rpcnum; /* RPC number */
  int argc; /* number of input arguments */
  void *argv[20]; /* argument array, maximum is 20 */
} PARRPC;

PARRPC *rpcin[MAXRPCS];

char *RpcNames0[] = {
  "find_a_cal",
  "find_all_cal",
  "add_a_cal",
  "del_a_cal",
  "change_a_cal",
  "Shutdown",
};
```
Chapter 2. Tools for Rapid Prototyping

```c
char **RpcNames[NumOfServers] = {
  RpcNames0,
};

/*
 * Some utility definitions and functions are omitted here.
 */

/* get server location by interface, take the 1st one */
static handle_t GetServerLoc(serNo)
int serNo; /* server number */
{
  lb_$lookup_handle_t ehandle = lb_$default_lookup_handle;
  static lb_$entry_t locs[5]; /* maximum locations: 5 */
  static unsigned long n_locs; /* # of locations */
  static int loc_i = 0; /* current index */
  status_$t st;
  handle_t rh;

  loc_i = 0;
  lb_$lookup_interface(&SerInterface[serNo], &ehandle, 5, &n_locs, locs, &st);
  if (st.all != status_$ok) {
    fprintf(stderr, "Cannot locate server - %s\n", error_text(st));
    return (handle_t) 0;
  }

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

  return rh;
}

/* Rpc "find_a_cal" call function of "ca" */
int ca_find_a_cal_func(rpcnum, argc, argv)
int rpcnum;
int argc;
void *argv[];
{
  handle_t rh;
  status_$t fst;
  pfm_$cleanup_rec crec;
  char name[MAXNAMELEN];
  char start[MAXVALLEN];
  static DISCAL_REC rec;

  /* get the server location */
  /* for saving time, the SerHandle[0] can be used */
  rh = GetServerLoc(0);
  if (rh == 0) return -1; /* cannot get location */

  fst = pfm_$cleanup(crec);
  if (fst.all != pfm_$cleanup_set) {
    if (fst.all != status_$ok)
```

Chapter 2. Tools for Rapid Prototyping

fprintf(stderr, "Exception raised in find_a_cal -
%sn", error_text(fst));
switch (fst.all) {
    case rpc_$comm_failure:
        fprintf(stderr, "Communication error\n");
        break;
    case rpc_$wrong_boot_time:
        fprintf(stderr, "Server fails\n");
        break;
    default:
        fprintf(stderr, "Unknown error\n");
        break;
}
    rpc_$free_handle(rh, &st);
    return -1;
}
/* converting "in" parameters */
conv_str(name, argv[0]);
conv_str(start, argv[1]);
/* actually call the remote procedure */
ca$find_a_cal(rh, name, start, rec);
printf("find_a_cal returned\n");
/*
return parameter checking here (omitted)
*/
    pfm_$rls_cleanup(crec, fst);
    return 0;

/*
Other remote procedure call frames are similar to the above, and are omitted here.
*/

/"ca": Rpc "find_a_cal" call use function */
static void ca_find_a_cal_use(rpcnum, argc, argv)
int rpcnum;
int argc;
void *argv[];
{
    if (ca_find_a_cal_func(rpcnum, argc, argv) == -1) {
        fprintf(stderr, "Cannot communicate with server -ca-\n");
    }
}
/*
Other remote procedure call frames are similar to the above, and are omitted here.
*/
void (*RpcUseFuncs0 []) () = {
    ca_find_a_cal_use,
Chapter 2. Tools for Rapid Prototyping

ca_find_all_cal_use,
ca_add_a_cal_use,
ca_del_a_cal_use,
ca_change_a_cal_use,
ca_shutdown_use,
0
}

void (**RpcUseFuncs [NumOfServers]) () = {
RpcUseFuncs0,
};

/*
Some parallel execution-related functions are omitted here.
*/

static void parRun(num)
/* parallel execution rpcs */
int num;  /* number of RPCs */
{
    int sernum, rpcnum, argct, i;
    if (num <= 0)
        return;
    if (num == 1) {
        sernum = rpcin[0]->sernum;
        rpcnum = rpcin[0]->rpcnum;
        argct = rpcin[0]->argc;
        (**RpcUseFuncs[sernum][rpcnum])(rpcnum, argct, rpcin[0]->argv);
        return;
    } else {
        /* 1<num<=NumOfRpcs */
        /* implemented calling format of a parallel RPC call */
        cobegin(num);
        for (i=0; i<num; i++)
            execut2(rpcin[i]->sernum, rpcin[i]->rpcnum,
                rpcin[i]->argc, rpcin[i]->argv);
        coend(num);
    }
}

/*
Some input-related functions are omitted here.
*/

/* main function */
main(argc, argv)
int argc;
char *argv[];
{
    int prpc, i, j, k;
    status_#t fst;
    pfm_#cleanup_recrec;
Chapter 2. Tools for Rapid Prototyping

allocPar(); /* address allocation */
if (argc < 2) usage(-1);

prpc = atoi(argv[1]);  /* Number of parallel RPCs */
for (i=0; i<prpc; i++)
  /* obtain the ith RPC info into rpcin[i] structure */
  if (getDetail(i) == -1) exit(0);

/*
Some functions that verify the testing message are omitted here.
*/

  /* set exception segment */
  fst = pfm_$cleanup(crec);
  if (fst.all != pfm_$cleanup_set) {
    if (fst.all != status_$ok)
      fprintf(stderr, "*** Exception raised - %s\n", error_text(fst));
      pfm_$signal(fst);
  }

  parRun(prpc);
  pgm_$exit();
}

The include file for both server and client drivers contains the necessary include files for general processing and RPC-oriented processing. It also contains some of the definitions obtained from the query or server definition file. The include file is indicated in Listing 2.6.

When using the make utility to construct the executable files, the NIDL compiler will generate the following files:

Table 2.2. Files generated by NIDL compiler

<table>
<thead>
<tr>
<th>Name</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca.h</td>
<td>include file generated by NIDL</td>
</tr>
<tr>
<td>ca_cstub.c</td>
<td>client stub file generated by NIDL</td>
</tr>
<tr>
<td>ca_cswitch.c</td>
<td>client switch file generated by NIDL</td>
</tr>
<tr>
<td>ca_sstub.c</td>
<td>server stub file generated by NIDL</td>
</tr>
</tbody>
</table>
Listing 2.6. Include file for calendar server ca.def

/* Include file of server "ca" */
/* Must be included by both clients and server drivers */

#ifndef MSDOS
  #include <sys/types.h>
  #include <signal.h>
#endif
#include <stdio.h>
#include <math.h>
#include "base.h"
#include "rpc.h"
#include "socket.h"
#include "lb.h"
#if defined apollo
  #include "/sys/ins/pfm.ins.c"
#else
  #ifndef MSDOS
    #include <setjmp.h>
    #define setjmp(buf) _msdos_setjmp(buf)
    #define longjmp(buf, val) _msdos_longjmp(buf, val)
    #extern long _msdos_setjmp(jmp_buf);
  #endif
  #include "u_pfm.h"
#endif
#if defined apollo
  #include "/sys/ins/task.ins.c"
#else
  #include "ca.h"
  #ifdef vax
    #include "vax.h"
  #endif
#endif
extern char *error_text();
extern uuid_t uuid_nil;
extern boolean rpc_debug, rpc_lossy;
int MaxCalls = 5;
int Family;

struct {
  socket_addr_t sa;
    unsigned long len;
} Loc;
char RegName[80] = "calendar server interface";
Finally two executable files will be created. They are: `ca_ser.r` (linked with `ca_ser.o` and `ca_sstub.o`), which is the server executable file, and `dp.r` (linked with `dp.o`, `ca_cstub.o`, and `ca_cswtch.o`), which is the client executable file. Because there are five remote procedures (not counting the shutdown procedure here), it is possible to execute them sequentially and concurrently. That is, the user can test this through the client driver.

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

Name: `add_a_cal`;

```c
/* record for adding into the database */
Param: in: DISCAL_REC rec;
/* number of existing records */
Param: out: int existing_num;
```

That is, one more parameter is added to the remote procedure. Now by running the generator again, the new NIDL file the appropriate segment will be changed to:

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

In the new RPC template the new segment will be:

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

    /* The contents of the RPC function */
}
```
Finally in the client driver, one of the changed segments is the remote procedure calling function.

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

By the help of the generator, the programmer can build the prototype programs of the distributed frame for the proposed distributed application in a very short time. Of course the distributed frame is of no use if real jobs cannot be performed in the RPC template file (in that case, the testing is also of no sense) or if a suitable user interface cannot be employed. The next two sections will address some aspects of the application modules and user interface prototyping.

2.4. User Interface Generator

2.4.1. Introduction

User interface prototyping has been discussed by several researchers. Christensen and Kreplin [Christensen84] described a user interface prototyping system which can generate prototypes from specification files in a dialogue format. In a specification file, the dialogue structure is defined. Athur [Athur87] described a formal menu-based system. A hierarchical structure is defined in his menu specification.

Intuitively, a menu system provides users with a set of selections and for each selection made executes an associated action. The selected action can be a function, or can be another menu. So the execution cycle is:
Chapter 2. Tools for Rapid Prototyping

(0). Initialise the main (first) menu to the current menu

(1). Display the current menu

(2). Obtain the user response

(3). If the response is not valid go to (2)

(4). If the selected selection is a function, execute it; go to (1)

(5). If the selected selection is a child menu, set it to the current and go to (1)

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

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

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

Let \( r(c_i, a_j) = (c_k, p_i) \). If \( a_j \) is not the exit to parent menu function, then we say that \( c_i \) is a parent menu of \( c_k \) and \( c_k \) is a child menu of \( c_j \). A history of a menu \( c \) is its parent path from the main menu (the root) to \( c \). Sometimes a user may wish to know the history of a current menu, especially when the menu path is too deep.
Chapter 2. Tools for Rapid Prototyping

2.4.2. Syntax

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

Listing 2.7. Interface definition file syntax

```
MENU ::= menu variable
         MENU_DEFS
         end menu variable

MENU_DEFS ::= MENU_DEF { ; MENU_DEF }

MENU_DEF ::= menu integer
            HEAD_DEF ;
            SEL_DEF
            end menu integer

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

DIR ::= direction: D_PARAM

D_PARAM ::= horizontal | vertical

POS ::= position: P_PARAM

P_PARAM ::= default
         | u_row = integer, u_col = integer, l_row = integer, l_col = integer

EXE ::= execution: FUNC

FUNC ::= variable()

COLOUR ::= foreground = integer, background = integer

SEL_DEF ::= selections:
            ONE_SEL { ; ONE_SEL }
            end selections

ONE_SEL ::= SEL_ITEM , HELP_ITEM

SEL_ITEM ::= string , CLASS

CLASS ::= child menu integer | function FUNC

HELP_ITEM ::= help string
```

The non-terminals variable, integer, and string have usual meanings as in the C programming language.
2.4.3. Semantics

We call the integer following the menu in MENU_DEF (Listing 2.7) the menu number. The following notes indicate the semantic issues:

(1). All menu numbers must be distinct.

(2) Any child menu can have only one parent menu. That is, if a menu number is used as a child menu number within a menu selection, it cannot be used in any other menu selection.

(3). When the menu is used to generate command level interface instead of window-oriented interface, the position definition, direction definition and colour definition are not applicable.

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

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

The user interface generator is used to generate the window-oriented or command-oriented interface screen to accept the user commands. It reads an interface definition file and translates the file into three related files. They are: an include file which contains the definitions of the user interface, an interface drive program (called the menu driver) which can be used to drive the execution of the interface program, and a dummy interface function module which defines all the interface functions used in the menu driver in dummy format. The definition files are self-explanatory and the
generated files are ready to execute after compiling. Figure 2.5 indicates the input and output files of the user interface generator.

2.4.4. Implementation

After the processing, the menu include file will contain all the static menu definitions obtained from the interface definition file. The menu data structure is called TNODE, and is defined as in Listing 2.8.

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

Figure 2.6 illustrates the meaning of these pointers. For menu node $n$, its pare pointer points to menu node $a$ because $n$ is one of $a$’s sons. Its prev pointer points to menu node $b$ because $b$ is its left brother. $n$’s fson pointer points to the its first son menu, $d$, and its next pointer points to its right brother $c$. 

---

Figure 2.5. Input and output of the user interface generator
Listing 2.8. Menu data structure

typedef struct tnode { /* for holding menu info */
    int nnum; /* node number */
    int mnum; /* menu number */
    int scnt; /* count of sons */
    int bnum; /* brother number */
    char *mname; /* menu name */
    char *help; /* help information */
    int class; /* node class -- leaf/leaf */
    int mdir; /* menu direction */
    int bord; /* border type */
    int urow; /* row position of upper-left */
    int ucol; /* col position of upper-left */
    int lrow; /* row position of lower-right */
    int lcol; /* col position of lower-right */
    int fclor; /* foreground colour */
    int bclor; /* background colour */
    char *fname; /* function name -- leaf node */
    char **smlst; /* son menu name list */
    char **hplst; /* son help information */
    struct tnode *pare; /* parent menu */
    struct tnode *fson; /* first son menu */
    struct tnode *prev; /* previous menu */
    struct tnode *next; /* next menu */
} TNODE;

Figure 2.6. Menu-tree pointers
Listing 2.9. Interface definition file example

/* user interface definition file for calendar */
/* (for testing) */

menu window-oriented

menu 0
  direction: horizontal;
  position: default;
  selections:
    "MEETING", child help "Enter a new meeting."
    "REMINDER", child help "Enter a new reminder."
    "SHOW", child help "Show my engagement."
  end selections
end menu 0;

menu 1
  direction: vertical;
  position: default;
  selections:
    "Create", function create()
    "Edit", child menu 4,
    "Modify existing meetings"
  end selections
end menu 1;

/*
definitions for menu 2, 3, ... etc, are defined here
*/

end menu window-oriented

Listing 2.9 provides the interface definition file for our distributed calendar system. Because we only want to show its main features, we list only two segments of the file.

Then, the generator will produce a menu driver program as indicated in Listing 2.10 and an include file as indicated in Listing 2.11.
Listing 2.10. Menu driver file

```c
#include "keys.h"
#include "menuprog.h"
#include "cal.h"

#include <stdio.h>
#include "s_menu.h"
#include "msckey.h"

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

menu_O(np)
TNODE *np;
{
    int i, cn, first_time;
    TNODE *snp;
    cn = 0;
    first_time = YES;
    while (1) {
        i = menu_subs(np, first_time, cn);
        first_time = NO;
        switch (i) {
            case 0:
                snp = find_son_i(np,i);
                cn = menu_1(snp);
                break;
            case 1:
                snp = find_son_i(np,i);
                cn = menu_2(snp);
                break;
            case 2:
                snp = find_son_i(np,i);
                cn = menu_3(snp);
                break;
            case EXIT:
                return (np->bnum + ESCAPE);
            default:
                break;
        }
    }
}
```
Chapter 2. Tools for Rapid Prototyping

```c
menu_1(np)
TNODE *np;
{
    int   i,cn,first_time;
    TNODE *snp;

    cn = 0;
    first_time = YES;
    while (1) {
        i = menu_subs(np, first_time, cn);
        first_time = NO;
        switch (i) {
            case 0:
                create();
                break;
            case 1:
                snp = find_son_i(np,i);
                cn = menu_4(snp);
                break;
            case LEFT:
            case RIGHT:
                return upper_choice(np,i);
            case EXIT:
                return (np->bnum + ESCAPE);
            default:
                break;
        }
    }
}

/*
other program segments are omitted here
*/

main(argc,argv)
int  argc;
char **argv;
{
    menu_0(menu_tree_node);
}```
Listing 2.11. Menu include file

```c
char *menu_0_name_list[] = {
    "MEETING",
    "REMINDER",
    "SHOW",
    0
};

char *menu_0_help_list[] = {
    "Enter a new meeting.",
    "Enter a new reminder.",
    "Show my engagement.",
    0
};

char *menu_1_name_list[] = {
    "Create",
    "Edit",
    0
};

char *menu_1_help_list[] = {
    "Input new meetings",
    "Modify existing meetings",
    0
};

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

TNODE menu_tree_node[] = {
    { 0, 0, 3, 0, "MAIN", 0, 1, 0, 1, 0, 0, 2, 79, 0,
        7, 0, menu_0_name_list, menu_0_help_list, 0,
        &menu_tree_node[1], 0, 0 },
    { 1, 1, 2, 0, "MEETING", 0, 1, 1, 1, 2, 4, 0, 0, 0,
        7, 0, menu_1_name_list, menu_1_help_list,
        &menu_tree_node[0], &menu_tree_node[2],
        0, &menu_tree_node[7] },
    { 2, -1, 0, 0, "Create", 0, 0, 0, 0, 0, 0, 0, 0,
        0, 7, "create()", 0, 0,
        &menu_tree_node[1], 0, 0, &menu_tree_node[3] },
    { 3, 4, 3, 1, "Edit", 0, 1, 1, 1, 2, 43, 0, 0, 0,
        7, 0, menu_4_name_list, menu_4_help_list,
        &menu_tree_node[1], &menu_tree_node[4],
        &menu_tree_node[2], 0 },
    { 7, 2, 2, 1, "REMINDER", 0, 1, 1, 1, 2, 18, 0, 0, 0,
        7, 0, menu_2_name_list, menu_2_help_list,
        &menu_tree_node[0], &menu_tree_node[8],
        &menu_tree_node[1], &menu_tree_node[10] },
    { 10, 3, 2, 2, "SHOW", 0, 1, 1, 1, 2, 30, 0, 0, 0,
```
The generator also produces a dummy interface function module. The format for each function of the interface function module is similar. Following is the dummy module for function `create()`:

```c
#include <stdio.h>

create()
{
    printf("This is function create(). \n");
}
```

The menu screens

![Diagram](Image1)

Figure 2.7. The menu screens
If the "WINDOW" option is set in the command when generating the interface, then the interface program will first display the menu as Figure 2.7(a). If the user selected MEETING, the menu screen will change to Figure 2.7(b). When Create is selected, the associated function create() will be executed.

If the "COMMAND" option is set in the command when generating the interface, then the interface program will display at first the menu as:

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

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

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

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

If there is any changes in the interface definition, the developer can simply change the definition file and re-generate the source files.

2.5. Application Module Generator

2.5.1. Introduction

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

Database management is one of the important issues in distributed IS application. A lot of database systems are available today which provide convenient interfaces to upper level programs. We selected two of them to be our underlying database systems. The first is the Apollo’s Database Access Manager [Perry89] which has very good performance in lightweight database applications and has been used in many distributed applications. The other is the BTree system [Softfocus88], which is highly portable, easy to use and also has been used in many distributed IS applications.

2.5.2. Syntax

A database description file defines the fields of each database file, the screen layout of each database file, and operations to be performed on these database files. The application generator generates the database files and program modules for database operations and screen layouts. The formal description of the database definition file is indicated in Listing 2.12.

2.5.3. Semantics

A database description file consists of a file definition part and an operation definition part. The optional \textit{CONSTS} and \textit{STRUCTS} parts are the same as in server definition file described in section 2.3.2. One or many database files can be defined in a single description file (this may make the joint operations possible). Each database file
Chapter 2. Tools for Rapid Prototyping

Listing 2.12. Database definition file syntax

DATABASE ::= $define variable
            [ CONSTS ]
            [ STRUCTS ]
            FILE ; { FILE ; }
            OPERATIONS
            $end define variable

FILE ::= $file variable
       FIELD { ; FIELD }
       $end file variable

FIELD ::= FD_NAME , TYPE , LENGTH , PROMPT , TEMPLATE ,
         PRMPT_ROW , PRMPT_COL , FIELD_ROW , FIELD_COL
         [, key ]

FD_NAME ::= variable
TYPE ::= TP [ JUSTIFYING ]
TP ::= A | C | D | N
JUSTIFYING ::= R | L

LENGTH ::= integer
PROMPT ::= string
TEMPLATE ::= string

PRMPT_ROW ::= integer
PRMPT_COL ::= integer
FIELD_ROW ::= integer
FIELD_COL ::= integer

OPERATIONS ::= operations
              OPERATION { ; OPERATION }
              $end operations

OPERATION ::= OP_NAME , OP , FILE_NAME

OP_NAME ::= name : variable
OP ::= ADD | CHG | DEL | FDA | FDG
FILE_NAME ::= variable

definition lists all its field names, types of the fields, lengths of the fields, prompts and templates used, and printing positions for the prompts and templates. If key is specified within a field, then the field is to be used as an index field for the database file. Table
Chapter 2. Tools for Rapid Prototyping

2.3 defines some field types.

Table 2.3. Field type definition

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

A field can be in right or left justified format by specifying its `JUSTIFYING` definition (it is unjustified by default).

The operation definition part defines all the basic operations over the defined database files. At this moment we have only implemented several simple operations over a single database file, such as add an entry, change an entry, delete an entry, etc. Complex operations such as joint operations are designed but not implemented.

The input and output files of the application module generator can be indicated as in Figure 2.8. From the definition file, the generator produces an application driver program, a database operation module, a screen layout operation model and some include files.

![Figure 2.8. Input and output files of the application module generator](image)
2.5.4. Implementation

The following listing provides the database definition file for our calendar database system. Please notice that the constant definition and data structure definition are the same as in server definition file case. So they are actually in an include file and are included by both definition files. The syntax in Listing 2.12 applies to the file after this inclusion has been performed.

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

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

The user interface generator is used by the application modules generator to produce the listing and selections for the operations defined in the database definition file. The fields of each database are displayed on the screen according to the definition file. Because our emphasis here is the distributed frame generation, this part of the model is currently incomplete.
Listing 2.13. Database definition file example

```plaintext
$define DISCAL
$file ca

MAXIDLEN = 20;
MAXNAMELEN = 20;
MAXVALLEN = 13;
MAXCMTLEN = 255;
MAXDATE = 6;

typedef struct {
    char mtID[MAXIDLEN]; /* meeting ID */
    char username[MAXNAMELEN]; /* user name */
    char start[MAXVALLEN]; /* meeting start time */
    char end[MAXVALLEN]; /* meeting end time */
    char caller[MAXNAMELEN]; /* meeting caller */
    char comment[MAXCMTLEN]; /* memo of the meeting */
    char issue_d[MAXDATE]; /* date of issuing */
} DISCAL_REC;

mtID, A, MAXIDLEN, "Meeting ID: ",
"[ ]", 3, 1, 3, 21, key;
username, A, MAXNAMELEN, "User name: ",
"[ ]", 4, 1, 4, 21, key;
start, A, MAXVALLEN, "Start date & time: ",
"[ ]", 5, 1, 5, 21, key;
end, A, MAXVALLEN, "End date & time: ",
"[ ]", 6, 1, 6, 21;
caller, A, MAXNAMELEN, "Caller: ",
"[ ]", 7, 1, 7, 21, key;
comment, A, MAXCMTLEN, "Comment: ",
"[ ]", 8, 1, 8, 21;
issue_d, A, MAXDATE, "Date of issuing: ",
"[ -/ / ]", 9, 1, 9, 21;
$end file ca;

$operations
    add_a_cal, ADD, ca;
    del_a_cal, DEL, ca;
    change_a_cal, CHG, ca;
    fund_a_cal, FDA, ca;
    find_all_cal, FDG, ca;
$end operations

$end define DISCAL
```
2.6. Interfacing the Generators

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

As indicated in Figure 2.9, three connections are needed. In the user interface and distributed frame connection, the user interface will use the remote operations provided by the distributed frame, but the distributed frame usually does not use functions from the user interface. So an include file is used to provide the user interface with all available remote procedure calls exported by all related servers from the client viewpoint.

As indicated in Figure 2.9, three connections are needed. In the user interface and distributed frame connection, the user interface will use the remote operations provided by the distributed frame, but the distributed frame usually does not use functions from the user interface. So an include file is used to provide the user interface with all available remote procedure calls exported by all related servers from the client viewpoint.

As indicated in Figure 2.9, three connections are needed. In the user interface and distributed frame connection, the user interface will use the remote operations provided by the distributed frame, but the distributed frame usually does not use functions from the user interface. So an include file is used to provide the user interface with all available remote procedure calls exported by all related servers from the client viewpoint.

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

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

### 2.7. Prototyping the Distributed Calendar System

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

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

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

At first we identified the following concepts:

- **Caller.** A person that issues the meeting.
- **Callee.** One of the persons that a caller wants to have a meeting with.
- **Meeting data.** A message which includes the caller’s name, the callee’s name, meeting time interval, and a brief description of the meeting purpose (see the Listing 2.2 in Section 2.3.3.).
- **Calendar database.** If a user has a meeting, there will be an entry in his database which contains the meeting data.
- The process of reminders is the same as for calendars except that the meanings of the fields are changed to the user’s name, the reminder’s time, and the reminder
Based on these assumptions and requirements, the first prototype was developed and tested by using our prototyping tools. It consisted of the following parts:

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

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

A research group manager and a sales person acted as users (the user). After executing the prototype together with the user, it is identified that there should be a meeting room database because room conflicts are also a problem. That is, two meetings cannot be assigned to the same meeting room with intersecting time intervals. So a room database was added, and a new field meeting_room was added into the calendar server definition file. That caused the distributed frame prototype be re-generated to have the meeting room server working co-operatively with the calendar server. Also the new client program can now use both databases simultaneously. Several new modules were added into the user interface prototype to fit the new database; and some new application modules were added. Also, the first requirement was changed to:
Chapter 2. Tools for Rapid Prototyping

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

From the first prototype to the second prototype, most changes were performed by re-writing the various definition files and only a small amount of code re-writing was needed.

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

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

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

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

When the system was put in use by some users, they found that there should be some restrictions on the issuing of meetings. That is, some priorities are needed among all users. So, the system was changed again that each user was assigned a priority number. In that case, only the users with higher or equal priorities can arrange meetings with users having lower or equal priorities. Because there had been a user information server existing in the LAN, an interface with that server was built and two fields were added into the database managed by the user information server. These two new fields specify the user priority and the user calendar server location, respectively. At last, the system was ready for use.
Not surprisingly the computer-aided rapid prototyping method will reduce the distributed IS application development cost. For example, in a reported prototyping experiment [Boehm84], systems were developed at 40% less cost and 45% less effort than conventional methods. Other researchers have reported even more impressive figures [Hekmatpour88]. As in our example, only one man week was used for the initial prototype while conventional methods were estimated to have required 4 man weeks.
3.1. Introduction

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

Sophisticated debugging tools are urgently needed. Unfortunately, few distributed debuggers are available today to support the development of distributed applications [Cheung90], especially in the area of RPC-oriented distributed programming. Most existing distributed debugging tools and models are based on the message passing model of process interaction, in which the send and receive primitives are used in
communications. The RPC-oriented programs, which are our main concern in this dissertation, have a different view of process interaction. In these programs, when a RPC call is issued, we want to know that how the relevant server responds to the call and how the returned results are accepted by the caller. That is, issuing a RPC call, receiving and processing the call, returning the call, and receiving the results of the call are the important events in a RPC program. There is no need to investigate how the calling message is sent from the caller to the callee and how the underlying communication primitives are used.

There are two major approaches to the debugging of distributed programs, namely, *debugging with repeated execution* of the program (or *cyclic debugging*) [McDowell89], and *debugging with trace* of program execution [Miller88]. In cyclic debugging, a user executes the program in a controlled manner until an error is detected. The program can be re-executed to produce the same execution behaviour. This is a very convenient way for small programs or programs that have little interaction between their concurrent parts. For a larger distributed program, however, executing the program several times while repeatedly setting breakpoints may be very costly. Also, sometimes the re-execution of a distributed program may not result the same behaviour because of the nondeterministic characteristics of the program. In debugging with trace, no reproducible behaviour is needed. The generated trace is deterministic and can be analysed in any controlled way that a programmer determine. The generation of the trace may, however, be very costly in time and space, and the events in the trace may not be fully ordered. Techniques are needed to analyse the program trace. In this chapter, the latter method is supported.

Usually a debugger based on tracing is called a monitor. In the following, the terms debugger and monitor are used interchangeably.

*Event–based* distributed debugging has been widely discussed. Bates and Wileden [Bates83] used a method called *behavioural abstraction* to hierarchically define high
level events in terms of sequences of primitive events (such as process creation, page fault, and message exchanges). The debugger monitors and traces these high level events. Only a simple display of these events is performed by their implementation. In a later work [Bates89], Bates described a system (EBBA) for debugging heterogeneous distributed systems based on his behaviour abstraction. EBBA builds models of program behaviour from system activity by recognising complex events in a program execution according to user specified hierarchical templates. The debugger can efficiently support distributed debugging by distributing itself in a distributed system. The main shortcoming of EBBA is that it costs an executing program 10-20ms to log a primitive event. Lin and LeBlanc [Lin89] describe the design of a debugger that provides event-based debugging of object/action programs in an object-oriented operating system called Clouds. Two phases are used in the debugging. During a monitoring phase, the debugger builds event logs of object invocations, while in an examination phase, the logged events will be reviewed or even re-executed.

In a distributed system, the re-execution of a long program can be very costly. In that case, it is important for a debugger to allow a user to examine the course of an erroneous execution without re-executing the program. Replay is one of the main techniques used in that purpose. LeBlanc and Robbins [LeBlanc85] provided some degree of replay in their debugger. After the collection of all events, the events are displayed sequentially. Both single step and continuous display are supported. The debugger is used for a particular language called Pronet [LeBlanc88] rather than for a general-purpose language. Garcia and Berman [Garcia85] used Petri nets to analyse stored traces and provided a sophisticated replay tool for concurrent system debugging. They stored some "important" events into the trace of a program’s execution. The replay mechanism uses these samples together with static symbol table information to produce a "movie" of the program execution.

In order to more effectively support the debugging process, operating system
Chapter 3. Debugging RPC Programs

Modification and hardware support are used by some researchers. Miller et al [Miller86] used a model of computation and measurement to implement a program monitor for the UNIX BSD 4.2 operating system. Some changes of the Berkeley UNIX kernel are involved in their implementation, so there is no insertion of extra statements into the programs being monitored nor the recompilation of programs. Marinescu et al [Marinescu89] presented a model for parallel and distributed software debugging in which they defined the intrusive and non-intrusive monitoring of program events. For intrusive monitoring, some process state changes or event ordering changes are induced by the debugger, whereas in non-intrusive monitoring no such changes are induced in the states and event ordering. It is evident that without the help of hardware instrumentation, a distributed debugger is always intrusive because a non-intrusive debugger must not share resources with the programs being debugged. Unfortunately, modifying the operating system kernel or having dedicated processors for debugging is not feasible for most researchers. So, most existing distributed debuggers are actually intrusive. What can be done for these debuggers is to reduce the interference to the process state and event ordering as much as possible.

Lazzerini and Prete [Lazzerini88] introduced the concept of a compound event, which is expressed in terms of either an accumulated event, a sequential event conjunction, a logical event disjunction or an instantaneous event conjunction. On the occurrence of an event, possible actions, such as trace and break traps, event counting and measurement of time interval, are taken. Cheng and Wallentine [Cheng89] used a knowledge-based language to specify the events in distributed debugging. In their language, an aggregate event can be constructed from primitive events and previously defined aggregate events. Any events can be monitored by the debugger.

We note that there have been many sequential debugging tools available and one can use them to debug each program part PP (see Section 1.3.3 for an explanation) of a distributed program separately. So we have the following two assumptions:
(1). The underlying hardware and software are correct. The only things that need to be debugged are the distributed application programs we have developed.

(2). The cooperation between the concurrent parts of a distributed program is the main focus in our debugging.

In this chapter we present a distributed monitor which has the ability to monitor RPC programs. Our debugger is event-based and we also have hierarchical event definitions, as with [Bates89] and [Lazzerini88]. We believe our definition of events has advantages in the expression of concurrent events, especially RPC calls. Our debugger can monitor any RPC program irrespective of whether it commences execution before the debugger or not (for example, a server program usually executes "forever"). The only requirement is that the program has to be linked with a debugging library. After debugging, the program can be re-linked with ordinary libraries for better performance. Also, no operating system modification is involved, and some techniques are used to reduce the interference of the monitor to a monitored RPC program. For example, our monitor costs an execution program less than 5ms (on average) to form a primitive event and insert it into the event queue. After that the execution program continues. This is an improvement over the EBBA system [Bates89] where the cost is 10-20ms.

Execution trace of a RPC program is stored in a distributed database and proper methods are used to replay the execution by using the information of the database. Our replay facility is better than LeBlanc and Robins [LeBlanc85] in that we allow simultaneously display of events of several concurrent processes.

3.2. Definitions and Assertions

We define two sets which will be used in the following description. We use $\Sigma$ to denote all possible events of a distributed program. Please note that usually a portion of the events in $\Sigma$ may occur during an execution of the program, Hence we use $E$ to
Chapter 3. Debugging RPC Programs

denote all the events occurred in a particular execution of the program. That is, we usually have $E \subseteq \Sigma$.

3.2.1. Primitive Event

In order to monitor events we have to assign each event a unique name.

**Definition 3.1.** An *event name* is a text object with the following syntax:

$$\text{eventname} ::= \text{TimeStamp} \cdot \text{SeqNo} \cdot \text{HostID} [ \cdot \text{affix }]$$

Here we use the same BNF notations as in Section 2.3.

The terminals have the following meaning:

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

If $en$ is an event name, we use $en.t$ to denote the timestamp, $en.r$ to denote the sequential number, $en.h$ to denote the host, and $en.a$ to denote the affix, respectively.

Because the affix component is optional, we need to achieve uniqueness with the first three components. Clearly, $en.h$ can be used to identify on which host the event happened, and $en.t$ denotes the occurrence time (relative to the local host) of the event. The use of the local clock register necessarily partitions (real) time into discrete intervals, as indicated in Figure 3.1. Here event $a$ and $b$ have the same timestamp $t_i$. 
Chapter 3. Debugging RPC Programs

Figure 3.1. Events with the same timestamp value

In the case that \( n \) events happened within one timestamp interval at the same host as above, \( en.r \) is used to distinguish these \( n \) events. We maintain an integer variable \( seq \) (called \textit{sequential number} variable) at each host. The variable is exclusively accessed by the \textit{TimeStamp} mechanism of the event generator, as indicated by (in C notation):

\[
a.r = (seq++) \mod N.
\]

The accessing of \( seq \), and its incrementing, is regarded as an atomic action. So if \( b \) happens at the same interval as \( a \) but the event generator accesses the sequential number variable after event \( a \), then \( a.r \) and \( b.r \) will have different values. We take \( N = 255 \), which gives 256 distinct values for the \( en.r \) field. Please notice that these sequential numbers cannot be directly used to order the events in the same timestamp interval. For example, event \( a \) may be assigned to value 255 while \( b \) be assigned to 0. We will discuss this matter in Section 3.4.1. The assignment and usage of the affix will be described in Section 3.3.5.

Another possible way to obtain unique event names is to let the \( seq \) be a sequential number register (say 32-bit) and use its sequential values as event identifiers. In this case the timestamp is unnecessary. But this approach has two shortcomings. First, the sequential numbers it generates are still limited. That is, if there are too many events to be named, this approach may breaks the naming uniqueness. Second, by using timestamps, one may have some feeling that how long is the time between two events. This may give some hints for debugging. But by using the sequential number register, it is impossible to obtain that feeling. So we think our approach is the right choice.
Assertion 3.1. In the above scheme, if the maximum number $n$ of events within a timestamp interval in any host satisfies $n \leq N + 1$, then all events in a distributed program are uniquely identified by using their event names.

Proof: It is evident that if two events of a distributed program occur in different hosts, then they will have different event names because of the HostID component of the event name. It is a basic assumption that host identifiers are unique through the distributed computer system. It is equally evident that if two events have a different timestamp they will have different names. So we need only consider the case where $n$ events occur in one host during the same timestamp interval. In that case, the host identifiers and timestamps of all these events are the same. The only way to distinguish them is through sequential numbers. From the generating mechanism, we know that these $n$ numbers are distinct. So we can certainly identify them within the local host. And then uniquely identify them in the whole distributed program.

Next we are going to define the primitive events used in our debugging. The definition is essentially intuitive.

Definition 3.2. A primitive event $e$ is defined as a pair $(f, m)$ where $f$ is a fact and $m$ is a message. In this context a fact is something which happens during a program’s execution and a message is some information associated to the fact.

A fact can be, for example, the creation of a process, the issuing and return of a RPC call, or an enquiring of a server’s location, and so on. A message can be, for example, the parameter values of a RPC call, the return information of a RPC return, the answer of a server location enquiring or even the process status.

Some kinds of relationship may exist between the primitive events of a distributed program. Two kinds of such relations are of particular interest in our context. They both involve at least two processes. The first kind is RPC-related relation and the second is the process-creation-related relation. We formalise them as follows:
Chapter 3. Debugging RPC Programs

(1). By \( rpc(x) \preceq y \) we mean that a RPC call relates events \( x \) and \( y \) such that either

(a). \( x \) is the "issuing a RPC call" event and \( y \) is the corresponding "begin execution of the remote procedure" event or,

(b). \( x \) is the "completion of remote procedure execution" event and \( y \) is the corresponding "RPC return" event.

By definition \( x \) and \( y \) are in different processes, or even in different hosts.

(2). By \( fork(x) \preceq y \) we mean that a process creation relates events \( x \) and \( y \) such that \( x \) is the "process creation" event and \( y \) is the corresponding "begin execution of the process" event.

In both cases we say that event \( x \) causes the occurrence of event \( y \), and denote as \( x \preceq y \).

**Corollary 3.1.** If \( x \preceq y \) is true, then \( y \preceq x \) is false.

**Proof:** \( x \preceq y \) means either \( rpc(x) \preceq y \) or \( fork(x) \preceq y \). If \( rpc(x) \preceq y \), then by definition, \( y \) is either "begin execution of the remote procedure," or "RPC return." So it is impossible to have \( y \preceq x \). If \( fork(x) \preceq y \), then \( x \) is the creation of a child process and \( y \) is the first event occurred in the child process. It is also impossible to have \( y \preceq x \).

Next definition is used to establish the partial ordering between events:

**Definition 3.3.** Let \( E = \{e\} \) be the set of all events of a RPC program's execution. For \( x, y : E \) we say \( x \) is the predecessor of \( y \) if and only if \( x \) and \( y \) satisfy one of the following conditions:

(1). \( x \preceq y \) (in different processes),

(2). \( x \) happened before \( y \) within the same process,

(3). \( x \) and \( y \) are in different processes, and there exist events \( a \) and \( b \) such that \( a \preceq b \) and \( x \) is a predecessor of \( a \) and \( b \) is a predecessor of \( y \), or

(4). \( x = y \).
Chapter 3. Debugging RPC Programs

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

**Assertion 3.2.** The pair $(E, \leq)$ is a partially ordered set.

**Proof:** A set is partially ordered set if it is reflexive, antisymmetric, and transitive [Alagar89]. Because we have defined $x \leq x$ for any $x : E$ (condition (4) of Definition 3.3), it is reflexive.

Suppose $x \leq y$ and $y \leq z, x, y$ and $z : E$. If $x, y$, and $z$ are in the same process, then by condition (2) of Definition 3.3 we have $x \leq z$.

If $x, y : P_1$ while $z : P_2$ (where $P_1$ and $P_2$ are event sets of different processes of a distributed program), then by condition (3) of Definition 3.3, there exist events $a$ and $b$ such that

$$y \leq a, \quad a \prec b, \quad \text{and} \quad b \leq z \quad (3.2.1)$$

If there is only one remote relation between $y$ and $z$, then we must have $a : P_1$ and $b : P_2$. So, by the condition (2), we have $x \leq a, a \prec b, \text{and} b \leq z$. That is, $x \leq z$.

Now, suppose the assertion holds for $n$ remote relations. If there are $n+1$ remote relations between $y$ and $z$ when (3.2.1) holds, then the number of remote relations between $y$ and $a$ (or equally, between $x$ and $a$) must be less than or equal to $n$, and so does the number of remote relations between $b$ and $z$. So we still have $x \leq z$. By induction, the assertion holds.

The proof for $x : P_1$ while $y, z : P_2$, and $x, y$ and $z$ are in three different processes are similar to the above. So the pair $(E, \leq)$ is transitive.

The antisymmetry can be proved by enumerating conditions of Definition 3.3. Let $x \leq y$ and $y \leq x$. Because $x \leq y$, $x$ and $y$ must satisfy one of the conditions in Definition 3.3. Suppose they satisfy the condition (1) of Definition 3.3, that is, $x \prec y$. Then $y \leq x$ must be false because by Corollary 3.1 $y \prec x$ is false and also $y$
and $x$ cannot satisfy other conditions.

Suppose $x$ and $y$ satisfy the condition (2) of Definition 3.3. Then they are in the same process and $x$ occurred before $y$. That means $y \leq x$ is false. The condition (3) is actually an extension of condition (1) and (2). So $x$ and $y$ also cannot satisfy condition (3). The only possibility for both $x \leq y$ and $y \leq x$ hold is that $x$ and $y$ satisfy the condition (4) of Definition 3.3. That is $x = y$. So $(E, \leq)$ is a partially ordered set.

3.2.2. Combined Event

Sometimes a user may be interested in the combination of several events. For example, if a server has two remote procedures that will access an object, it is interesting to see if these two procedures are all called during the execution, or to know the execution order of them. For defining events combination, we borrowed a notation from temporal logic [Kroger87], and define its meaning as follows:

- $\Diamond A$: Eventually operator. Means that there is a time point after the reference point (the present time) at which $A$ occurs.

We give the following definitions which can combine several events to form a new event.

**Definition 3.4.** Consider events $e_1, e_2 : \Sigma$, where $e_1 = (f_1, m_1)$ and $e_2 = (f_2, m_2)$. By $f_1 * f_2$ we mean that the fact part of $e_1$ is followed-by the fact part of $e_2$. By $f_1 + f_2$ we mean that the fact part of $e_1$ and the fact part of $e_2$ are independent. Similarly we define $m_1 * m_2$ as message $m_1$ followed-by message $m_2$ and define $m_1 + m_2$ to mean they are independent each other.

The definition is also valid for more than two facts (messages). But in our monitor, we have only implemented binary operations (see Section 3.3.5).
Chapter 3. Debugging RPC Programs

We do not give the details of followed-by and independent operations here. They will be discussed in Section 3.3.3, where we will define the actual data structure of an event and describe these two operations. What we can say up to this moment is that in followed-by operation, two or more facts (or equally, messages) are sequentially connected, while in independent operation, there is no such connection between these facts (messages).

Definition 3.5. Let $e_1, e_2 : \Sigma$ (please notice that they do not necessarily occur during one execution of a distributed program). We define:

\[
\begin{align*}
  e_1 * e_2 & \text{ if } (\bullet e_1 \text{ and } \bullet e_2) \text{ and } e_1 \leq e_2. \\
  e_1 + e_2 & \text{ if } (\bullet e_1 \text{ and } \bullet e_2) \text{ and } (e_1 \leq e_2) \text{ and } (e_2 \leq e_1). \\
  e_1 \cap e_2 & \text{ if } (\bullet e_1 \text{ and } \bullet e_2). \\
  e_1 \cup e_2 & \text{ if } (\bullet e_1 \text{ or } \bullet e_2). 
\end{align*}
\]

So, formula (3.2.2) means that both events occurred and they have $\leq$ relation. Formula (3.2.3) means that both events occurred, but there is no relation between them. Formula (3.2.4) simply means that the two events occurred. Finally (3.2.5) means that either one of the events occurred, or both of them occurred.

It is easy to see that

\[
(e_1 \cup e_2) \rightarrow (e_1 \cap e_2) \rightarrow (e_1 + e_2) \rightarrow (e_1 * e_2),
\]

and

\[
e_1 \cup e_2 = e_1 \cap e_2 \text{ or } e_1 \text{ or } e_2.
\]

Definition 3.6. If $e_1 = (f_1, m_1)$ and $e_2 = (f_2, m_2)$ are events, then

\[
\begin{align*}
  e_1 * e_2 & = (f_1 * f_2, m_1 * m_2), \\
  e_1 + e_2 & = (f_1 + f_2, m_1 + m_2), \\
  e_1 \cap e_2 & = (f_a, m_a), \quad \text{and} \quad e_1 \cup e_2 = (f_b, m_b).
\end{align*}
\]

are also events, where
Chapter 3. Debugging RPC Programs

\[
f_a = \begin{cases} 
  f_1 \cdot f_2 & \text{if } e_1 \leq e_2 \\
  f_2 \cdot f_1 & \text{if } e_2 \leq e_1, \\
  f_1 + f_2 & \text{otherwise}
\end{cases}
\]

\[
f_b = \begin{cases} 
  f_a & \text{if } e_1 \cap e_2 \\
  f_1 & \text{if } e_1 \text{ and } \neg e_2, \text{ and } \\
  f_2 & \text{if } e_2 \text{ and } \neg e_1
\end{cases}
\]

\[
m_a = \begin{cases} 
  m_1 \cdot m_2 & \text{if } e_1 \leq e_2 \\
  m_2 \cdot m_1 & \text{if } e_2 \leq e_1, \\
  m_1 + m_2 & \text{otherwise}
\end{cases}
\]

\[
m_b = \begin{cases} 
  m_a & \text{if } e_1 \cap e_2 \\
  m_1 & \text{if } e_1 \text{ and } \neg e_2, \text{ and } \\
  m_2 & \text{if } e_2 \text{ and } \neg e_1
\end{cases}
\]

We denote these new events as **combined events** and define \( e_1 \) and \( e_2 \) (or, \( e_1 \) or \( e_2 \), depends on both of them occurred or only one of them occurred) as their predecessors (also called **components** of the combined event). Combined events are usually defined by programmers, so they are of interest to users. For example, in Figure 3.2, there are two processes \( P_1 \) and \( P_2 \). The following relations among their events hold (of course, there are other relations which exist):

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

\[
\begin{array}{c|c|c}
\text{Process P1} & \text{Process P2} & \text{Time} \\
\hline
... & \cdots & \downarrow \\
\uparrow \downarrow a1 & \downarrow b1 & \downarrow \\
\uparrow a2 & \downarrow b2 & \downarrow \\
\downarrow a3 & \downarrow b3 & \downarrow \\
\downarrow \uparrow a4 & \downarrow b4 & \downarrow \\
\downarrow \quad \cdots & \downarrow b5 & \downarrow \\
\end{array}
\]

**Figure 3.2. Events relations**

It is easy to see that

\[
e_1 \cup e_2 \equiv e_2 \cup e_1; \ e_1 \cap e_2 \equiv e_2 \cap e_1; \ e_1 + e_2 \equiv e_2 + e_1.
\]
Chapter 3. Debugging RPC Programs

The priority of the above operators are, from high to low, $\cap$, $\cup$, $\ast$, $\ast$. So the expression $e_1 \ast e_2 \cap e_3 \cup e_4 + e_5$ is actually $(e_1 \ast ((e_2 \cap e_3) \cup e_4)) + e_5$.

Assertion 3.3. If $E$ is the set of all (primitive and combined) events of a RPC program’s execution, then $(E, \leq)$ is still a partial ordered set.

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

Definition 3.7. We call event $a$ the immediate predecessor of event $b$ if

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

For example, in Figure 3.2, $a_2$ and $b_4$ are immediate predecessors of $a_3$. But $a_1$ and $b_2$ are not.

3.3. Monitor Structure

3.3.1. Overview

The monitor consists of a controller and a group of managing servers. The controller has two main parts: a user interface (which incorporates a command interpreter and Input/Output functions) and a filter. A managing server consists of a server (MS server), an event queue and an event database. Each host which supports one or more monitored program parts has a managing server. The controller can be located at any host. By communicating with the associated servers, the controller can present the monitored results to the user. Figure 3.3 illustrates the structure of the monitor.

Debugging using this monitor system involves three stages or steps.

1. Monitoring. In the monitoring step, all events that occur on a particular host are monitored by the local MS and recorded in the local event database. All primitive
events including RPC calls and executions and process forks in both client and server program parts are monitored. Further a user can define combined events via *Event Definition File (EDF)* (described in Section 3.3.5). The monitor system is able to record such combined events.

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

![Figure 3.3. The structure of the distributed monitor](image)

(3). *Replaying*. By combining the results on all related event databases, the filter can present an execution trace of the distributed program to the user. The displaying speed and viewing contents are controlled by the user through the command...
Chapter 3. Debugging RPC Programs

interpreter. Two kinds of user interfaces are provided, namely, a command level interface, and a graphics interface. A user can use commands to order the monitor to start a remote/local program part and leave the monitor to monitor it. On the other hand, the user can monitor a program part which has been executing.

3.3.2. Debugging Library

There is no doubt that an effective distributed debugger has to be deeply embedded into the operating system [Harter85] to achieve sufficient speed and transparency. In order to monitor a program’s activity without causing any side-effects on its behaviour, operating system kernel modification or hardware support is essential. Because of the difficulty of modifying operating systems and providing hardware support, most of the debugger and monitor researchers use software techniques as the substitute. This makes the implementation much easier at the cost of efficiency, especially for real time systems. The performance may be completely unacceptable for real time programs. In this stage, we are not in a position to create hardware support, or to change the operating system kernel. Instead, we have provided a debugging library, which has to be linked with the program that is to be monitored. This library provides replacements for some of the operating system calls, such as fork. It also has some functions that replace the NCS-related calls and other service functions. Each replaced function does the following two things:

1. Event creation and queueing. It first forms the primitive event. Then inserts the event entry into the local event queue for the local managing server to process.

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

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

3.3.3. Managing Server

On each host, there is a managing server which consists of a server (MS) and an event database. Each event database is simply a set of entries where each entry records an event and has the structure indicated in Listing 3.1.

Listing 3.1. Event data structure

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

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

The `name`, `pid`, `prg_name`, `p_name`, and `s_name` fields are self-explanatory. The fact information (`fact_info`) of an event is a character string that provides a readable description of the fact. For primitive events, they are assigned by the debugging library. For example, they may be "begin RPC call (RPC name)" or "fork new process." For combined events, they are assigned by programmers (see Section 3.3.5). The message part is stored as a string of bytes and a piece of type information. The filter uses the type information to interpret the byte string and to display the result to the
Chapter 3. Debugging RPC Programs

user through the command interpreter.

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

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

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

A MS server has the following functions:

1. **Database management.** It is responsible for the management of the local event database. The database is protected by the MS server and any access of it must go through the MS server.

2. **Event logging.** When an event occurs, the event name is built and it is put into a local event queue by the executing program (see Section 3.3.2). The local MS server is then responsible for logging it in the local event database. In this case, the monitored process can continue immediately after the event has been queued, instead of waiting for the local MS server to insert the event into the database.

3. **Communicating with the controller.** All communications among the servers are conducted by the controller. A lot of commands (see Section 3.3.4) are issued through the controller and performed by the server. This is the only way a user can access the event database.

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

### 3.3.4. Controller

The controller consists of a command interpreter and a filter. A list of main user commands follows (Table 3.1):

<table>
<thead>
<tr>
<th>Command</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>Monitoring</td>
<td>Use a Event Definition File</td>
</tr>
<tr>
<td>2</td>
<td>Invoke a PP for monitoring</td>
</tr>
<tr>
<td>3</td>
<td>Monitor an executing server PP</td>
</tr>
<tr>
<td>Ordering</td>
<td>Order the events of a process</td>
</tr>
<tr>
<td>2</td>
<td>Order the events of a PP</td>
</tr>
<tr>
<td>3</td>
<td>Order combined events</td>
</tr>
<tr>
<td>Replaying</td>
<td>Replay the execution of a PP</td>
</tr>
<tr>
<td>2</td>
<td>Replay the execution of a process</td>
</tr>
<tr>
<td>3</td>
<td>View combined events</td>
</tr>
<tr>
<td>4</td>
<td>Dump the trace of a PP</td>
</tr>
<tr>
<td>5</td>
<td>Dump the trace of a process</td>
</tr>
<tr>
<td>6</td>
<td>Dump combined events</td>
</tr>
<tr>
<td>7</td>
<td>Clean up trace of a PP</td>
</tr>
</tbody>
</table>

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

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

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

### 3.3.5. Event Definition File

All primitive events are automatically logged by the monitor if the monitored program is linked with the debugging library. Sometimes a user may find it is more convenient to define some high-level events and use them in debugging. An event definition file is used for that purpose. The syntax of the EDF specification language is indicated in Listing 3.2.

The following notes are relevant:

1. *Variable* has usual meaning as in the C programming language.
2. *Event_Affix* is the same as *affix* defined in Definition 3.1 (a character string).
3. Any *variable* that appears in the right hand side of an *EventExpression* must be the left hand side of an earlier *EventExpression*.
4. *Fact_info* is a character string as described in section 3.3.3.
5. *Inter* is the keyword for operation $\bigcap$ and *union* is the keyword for operation $\bigcup$. They are defined in Definition 3.4 to Definition 3.6.

We give a very simple EDF file example in (3.3.1). It defines a combined event *BothAccessed* as the intersection of two primitive events *AccessingDB1* and *AccessingDB2*. That is, if both primitive events occur, then the combined event is also deemed to occur.
Listing 3.2. Event definition file syntax

EDF ::= \texttt{BEGIN}  
\texttt{SPEC}  
\texttt{COMBINE}  
\texttt{END}  
\texttt{SPEC} ::= \texttt{PreSpec} ; \texttt{ComSpec} ;  
\texttt{PreSpec} ::= \texttt{Primitive Event:}  
Event\_Affix \{ , Event\_Affix \}  
\texttt{ComSpec} ::= \texttt{Combined Event:} \texttt{(variable, fact\_info)}  
\{ , \texttt{(variable, fact\_info)} \}  
\texttt{COMBINE} ::= \texttt{EventAssignment} \{ ; \texttt{EventAssignment} \}  
\texttt{EventAssignment} ::= \texttt{variable} = \texttt{EventExpression}  
\texttt{EventExpression} ::= \texttt{FFollow} | \texttt{EventExpression}  
\quad + \texttt{FFollow}  
\texttt{FFollow} ::= \texttt{FUnion} | \texttt{FFollow} * \texttt{FUnion}  
\texttt{FUnion} ::= \texttt{FInter} | \texttt{FUnion union FInter}  
\texttt{FInter} ::= \texttt{Operand} | \texttt{FInter inter Operand}  
\texttt{Operand} ::= \texttt{Event\_Affix} | \texttt{variable}  
\quad | \texttt{(EventExpression)}

\begin{verbatim}
BEGIN  
primitive Event:  
AccessingDB1, AccessingDB2;  

Combined Event:  
(BothAccessed, "Both DBs are accessed"); \hfill (3.3.1)  

BothAccessed = AccessingDB1 inter AccessingDB2;  
\texttt{END}
\end{verbatim}

Several steps are needed to use an EDF file. Firstly, the user inserts into the monitored program parts \textit{affix definition functions} before each primitive event which is to be used in the EDF file. The format of the affix definition function (defined in the debugging library) is
Chapter 3. Debugging RPC Programs

affix_define (affix),

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

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

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

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

\[ x = y \times z, \quad x = y + z, \quad x = y \cap z, \quad x = y \cup z. \]

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

3.4.1. Ordering Events

As we have pointed out, the local clocks of the hosts in the distributed system are not synchronised, so we cannot use the timestamp in definition 3.1 to order all events. According to our Assertion 3.2 and 3.3, there do exist some partial ordering among all events occurred during an execution. The following steps are used to establish the partial ordering among all events:

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

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

3. Combined event related predecessors. When an event with an affix definition occurs, it will cause the controller to evaluate the related combined event expressions, as described in Section 3.3.5. So, the name of the event is sent to the controller and stored as one of the immediate predecessors of the related combined events.
Chapter 3. Debugging RPC Programs

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

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

$$0 < 1 < 2 < \cdots < 254 < 255 < 0 < 1 < \cdots$$

(3.4.1)

As we have assumed that no more than 256 events occur between two timestamps, building the ordering among these events are straightforward. So, in one process, the immediate predecessor of event $y$ is event $x$ if $x.t$ is immediately less than $y.t$ and the (immediate) successor of $y$ is event $z$ if $z.t$ is immediately greater than $y.t$. If $e_1, e_2, \ldots, e_n$ are all events that have the same timestamp $t$ ($n \leq 256$), then the round ordering (3.4.1) is used to order them. This ordering process is performed by each MS server concurrently.

We use the example of Figure 3.4 to illustrate the essential aspects of above steps. In the picture, we have two processes $A$ and $B$. They execute on different hosts and their local MS servers are indicated $MS_A$ and $MS_B$, respectively. Suppose that $A$ makes a RPC call to one of $B$’s procedures. Let us use $a_1$ to represent event "issuing a RPC call" and $a_2$ the event "RPC return" for $A$. For $B$, we use $b_1$ to represent the event "begin execution of the remote procedure." and $b_2$ the event "completion of remote procedure execution." When $a_1$ occurs, $a_1$’s name is sent to $MS_B$ together with the
RPC call, and $MS_B$ will use this name as event $b_1$'s predecessor (that is, store this name together with event $b_1$). When $b_2$ occurs, that is, the remote procedure call returns, the event name $b_2$ is sent back to $MS_A$ together with the returning information and is stored as $a_2$'s predecessor. This was described in stage (1) above.

At this time, $a_1$ does not know that $b_1$ is its remote successor and $b_2$ does not know that $a_2$ is its remote successor. During stage (4), $MS_A$ finds that $a_2$ has a predecessor named $b_2$ and that $b_2$'s local MS server is $MS_B$. So $MS_A$ will send $(b_2, a_2)$ to $MS_B$ with the help of the filter and ask $MS_B$ to store $a_2$ as the remote successor of $b_2$. At the same time, $MS_B$ will discover that $b_1$ has a remote predecessor named $a_1$, and will ask $MS_A$ (with the help of filter) to store $b_1$ as $a_1$'s remote successor. At stage (4), the ordering between all $a_i$'s and $b_i$'s will be established by $MS_A$ and $MS_B$, respectively.

![Figure 3.4. RPC communication](image)

By using the above method, all RPC communication and process creation events can be partially ordered by predecessor/successor relation. For the events within a
Chapter 3. Debugging RPC Programs

single process, we can fully order them by their timestamps and sequential numbers. Combining with these two relationships together, we can have a partial ordering over all events of an RPC program. That makes the replay possible.

3.4.2. Process Replay

After the monitor has collected all the events of a RPC program’s execution, one can use the replay facility of the monitor to simulate the execution in controlled manner, and analyse the events in detail. As we know, a RPC program has several parts which may execute concurrently in different hosts. So, the replaying of a RPC program is to replay its program parts, sequentially or concurrently, on the controller’s terminal. For each program part, we can use the partial order relation to build an event graph and use this graph for replaying and analysis. The building process is simple. If \(a, b : E\) (\(E\) is the event set of the monitored distributed program), and \(a\) is a immediate predecessor of \(b\), then we have an arc from \(a\) to \(b\).

As we know, the only method that two program parts of a distributed program can communicate with each other is by RPCs. If we do not consider any RPCs at this moment, the event graph of each PP of a RPC program is then a weakly connected acyclic graph. It is easy to see that the event graph built by the predecessor relation is an acyclic graph. This graph is also weakly connected because the only way for a program part to branch out is by using process fork functions. The execution of these functions are primitive events and the predecessor relation between parent and child processes will be established by the monitor. So, if we consider the event graph as a non-directed graph, it will be a connected graph. The weak connection of the event graph follows.

Usually the client parts of a RPC program will communicate with all of its server parts by using RPC calls. In that case, the event graphs of all program parts of the RPC program will be connected by the predecessor relation of those RPC calls. So, usually
the event graph of a RPC program is also a weakly connected acyclic graph.

If we add the successor relation into the event set and build the graph, the result event graph will be a bi-directed graph. In that case, it is easy to control the event accesses during replay. That is, if the graph is weakly connected (this is the usual case), we can access any event from the beginning of the program, or even from any event. In our monitor, a user can control the event replay speed (or even single step), view any program part of a RPC program, view the details (such as messages and fact fields) of an event, and dump the results into files. The events can be displayed in both predecessor or successor orders. By using the graphics interface, one can display several event graphs on the controller terminal in several windows, and view the replays of these event graphs concurrently.

3.5. An Example

We use the distributed calendar database example described in Section 2.3.3 to illustrate our debugging process. Please note that the generated programs listed in Section 2.3.3 are modified and extended for real use. For example, all the remote procedures are implemented and additional features are added into the client program.

We first compile the server program and client program together with the debugging library. From Listing 2.4, we know the server at first registers itself to the location broker. Then it listens for client calls. If a call comes, the server calls the appropriate remote procedure and returns the result to the caller. After that, it listens to client calls again. If several client calls arrive, the server splits into several tasks (by using multi-programming tools instead of fork to processes [Apollo87]), with each task serving a call. As we know from Listing 2.3, there are six remote procedures (functions), as indicated in Table 3.2:
Chapter 3. Debugging RPC Programs

Table 3.2. Remote procedure calls of the calendar system

<table>
<thead>
<tr>
<th>Remote Procedures</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>ca$find_a_cal()</td>
<td>Find a meeting entry by name and start time</td>
</tr>
<tr>
<td>ca$find_all_cal()</td>
<td>Find all meeting entries for a user</td>
</tr>
<tr>
<td>ca$add_a_cal()</td>
<td>Add a meeting entry into the database</td>
</tr>
<tr>
<td>ca$del_a_cal()</td>
<td>Delete an entry from the database</td>
</tr>
<tr>
<td>ca$change_a_cal()</td>
<td>Change the contents of an entry</td>
</tr>
<tr>
<td>ca$shutdown()</td>
<td>Shut down the server</td>
</tr>
</tbody>
</table>

From Listing 2.5, we know the client first does some initialisation, and via a user dialogue determines which remote procedures of the server are to be called, as well as the input parameters of these calls. If there is only one call, the client issues the call and returns the result. If there are more than one call, these calls are made concurrently by the client.

Now, we perform the monitoring step. We let the server and client programs run in two different hosts and make two calls concurrently from the client. The two calls are ca$find_a_cal() and ca$add_a_cal(). Then the server program has the following events (as indicated in Table 3.3) recorded and some predecessor relations are also recorded (please refer to Step (1) and (2) of Section 3.4.1 for building these relations):

Table 3.3. Server events

<table>
<thead>
<tr>
<th>No.</th>
<th>Event name</th>
<th>Meaning</th>
<th>Process</th>
<th>Predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45978a307530.00.0282380905000000</td>
<td>Server begins</td>
<td>3958</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>45978a3186a0.01.0282380905000000</td>
<td>Listening</td>
<td>3958</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>45978a5947d8.02.0282380905000000</td>
<td>Enter ca$find_a_cal()</td>
<td>3958</td>
<td>C7</td>
</tr>
<tr>
<td>4</td>
<td>45978a5a6ee8.03.0282380905000000</td>
<td>Leave ca$find_a_cal()</td>
<td>3958</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>45978a5a8b88.04.0282380905000000</td>
<td>Enter ca$add_a_cal()</td>
<td>3958</td>
<td>C11</td>
</tr>
<tr>
<td>6</td>
<td>45978a5bc350.05.0282380905000000</td>
<td>Leave ca$add_a_cal()</td>
<td>3958</td>
<td>-</td>
</tr>
</tbody>
</table>

(We used simplified event names to denote predecessors. We use $Si$ ($i=1$ to 6) to denote server events and $Ci$ ($i=1$ to 12) to denote client events, where $i$ is the No. field of Table 3.3 and 3.4.)
Table 3.4. Client events

<table>
<thead>
<tr>
<th>No.</th>
<th>Event name</th>
<th>Meaning</th>
<th>Process</th>
<th>Predecessor</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>45978a441170.00</td>
<td>Client begins</td>
<td>13641</td>
<td>-</td>
</tr>
<tr>
<td>2</td>
<td>45978a5822e0.01</td>
<td>Prepare concurrent call</td>
<td>13641</td>
<td>-</td>
</tr>
<tr>
<td>3</td>
<td>45978a583668.02</td>
<td>Making concurrent call</td>
<td>13641</td>
<td>-</td>
</tr>
<tr>
<td>4</td>
<td>45978a5ccf90.11</td>
<td>End concurrent call</td>
<td>13641</td>
<td>-</td>
</tr>
<tr>
<td>5</td>
<td>45978a59b2f20.03</td>
<td>New process begins</td>
<td>13642</td>
<td>C3</td>
</tr>
<tr>
<td>6</td>
<td>45978a5aee0.07</td>
<td>Binding server</td>
<td>13642</td>
<td>-</td>
</tr>
<tr>
<td>7</td>
<td>45978a5b15f0.08</td>
<td>Calling ca$find_a_cal()</td>
<td>13642</td>
<td>-</td>
</tr>
<tr>
<td>8</td>
<td>45978a5cb330.10</td>
<td>ca$find_a_cal() returns</td>
<td>13642</td>
<td>S4</td>
</tr>
<tr>
<td>9</td>
<td>45978a59b2f20.04</td>
<td>New process begins</td>
<td>13643</td>
<td>C3</td>
</tr>
<tr>
<td>10</td>
<td>45978a59e630.05</td>
<td>Binding server</td>
<td>13643</td>
<td>-</td>
</tr>
<tr>
<td>11</td>
<td>45978a5a0d40.06</td>
<td>Calling ca$add_a_cal()</td>
<td>13643</td>
<td>-</td>
</tr>
<tr>
<td>12</td>
<td>45978a595f8.09</td>
<td>ca$add_a_cal() returns</td>
<td>13643</td>
<td>S6</td>
</tr>
</tbody>
</table>

The events of the client program as well as some predecessor relations recorded during monitoring are listed in Table 3.4. Please notice that the host ID is now different (server host is 0282380905000000 and client host is 0282380903000000), and there are two new processes created (each for a RPC call).

After we start the ordering step, the first thing we are going to do is to order all remote successors (see step (4) of Section 3.4.1). The result of remote predecessors and successors can be shown in Table 3.5. We also use the simplified event names.

Table 3.5. Remote predecessor/successor relations

<table>
<thead>
<tr>
<th>Event</th>
<th>Predecessor</th>
<th>Successor</th>
</tr>
</thead>
<tbody>
<tr>
<td>S3</td>
<td>C7</td>
<td>-</td>
</tr>
<tr>
<td>S4</td>
<td>-</td>
<td>C8</td>
</tr>
<tr>
<td>S5</td>
<td>C11</td>
<td>-</td>
</tr>
<tr>
<td>S6</td>
<td>-</td>
<td>C12</td>
</tr>
<tr>
<td>C3</td>
<td>-</td>
<td>C5, C9</td>
</tr>
<tr>
<td>C5</td>
<td>C3</td>
<td>-</td>
</tr>
<tr>
<td>C7</td>
<td>-</td>
<td>S3</td>
</tr>
<tr>
<td>C8</td>
<td>S4</td>
<td>-</td>
</tr>
<tr>
<td>C9</td>
<td>C3</td>
<td>-</td>
</tr>
<tr>
<td>C11</td>
<td>-</td>
<td>S5</td>
</tr>
<tr>
<td>C12</td>
<td>S6</td>
<td>-</td>
</tr>
</tbody>
</table>
The second thing now is to order events within each process (see step (5) of Section 3.4.1). Table 3.6 lists the final result of all predecessor and successor relations.

The last step is replaying. We use the relations listed in Table 3.6 to build a event graph (as show in Fig 3.5), and several commands are used to display the graph as well as the messages related to each displayed events. Please notice that we only displayed the predecessor relation in Figure 3.5 (using arrows). We are not going to describe the details here.

Table 3.6. Final predecessor/successor relations

<table>
<thead>
<tr>
<th>Event</th>
<th>Predecessor</th>
<th>Successor</th>
</tr>
</thead>
<tbody>
<tr>
<td>S1</td>
<td>-</td>
<td>S2</td>
</tr>
<tr>
<td>S2</td>
<td>S1</td>
<td>S3</td>
</tr>
<tr>
<td>S3</td>
<td>C7, S2</td>
<td>S4</td>
</tr>
<tr>
<td>S4</td>
<td>S3</td>
<td>C8, S5</td>
</tr>
<tr>
<td>S5</td>
<td>C11, S4</td>
<td>S6</td>
</tr>
<tr>
<td>S6</td>
<td>S5</td>
<td>C12</td>
</tr>
<tr>
<td>C1</td>
<td>-</td>
<td>C2</td>
</tr>
<tr>
<td>C2</td>
<td>C1</td>
<td>C3</td>
</tr>
<tr>
<td>C3</td>
<td>C2</td>
<td>C5, C9, C4</td>
</tr>
<tr>
<td>C4</td>
<td>C3</td>
<td>-</td>
</tr>
<tr>
<td>C5</td>
<td>C3</td>
<td>C6</td>
</tr>
<tr>
<td>C6</td>
<td>C5</td>
<td>C7</td>
</tr>
<tr>
<td>C7</td>
<td>C6</td>
<td>S3, C8</td>
</tr>
<tr>
<td>C8</td>
<td>S4, C7</td>
<td>-</td>
</tr>
<tr>
<td>C9</td>
<td>C3</td>
<td>C10</td>
</tr>
<tr>
<td>C10</td>
<td>C9</td>
<td>C11</td>
</tr>
<tr>
<td>C11</td>
<td>C10</td>
<td>S5, C12</td>
</tr>
<tr>
<td>C12</td>
<td>S6, C11</td>
<td>-</td>
</tr>
</tbody>
</table>
Chapter 3. Debugging RPC Programs

3.6. Remarks

The design of a RPC-oriented distributed program monitor is described in this chapter. Three steps are used to monitor a distributed program. In the monitoring step, the monitor’s managing servers record the events of the distributed program parts of their hosts and log them into their local event databases. In the ordering step, all events of a RPC program are partially ordered by using an ordering method and the event graphs of all program parts can be built. These event graphs are then used to replay the program's execution in the replaying step. Facilities are also provided to define combined events, to control the replay process, and to view the details of the message part of an event.

Our monitor is event-based and the event definition has advantages to express RPC-oriented events. We also made some efforts to reduce the interference of the monitor to the monitored RPC program.
Chapter 4
Execution Time Evaluation

4.1. Existing Models and Their Limitations

The remote procedure call is a powerful primitive for distributed programs, and the growing interest in its use demands tools for modeling and analysing the performance of such programs. Because a remote procedure call blocks the calling process until the call is completed and a reply has been received, a concurrency primitive such as COBEGIN or FORK is usually used to introduce the parallelism into the program. For example, in our distributed frame prototyping tools, we used COBEGIN–COEND primitives to execute several remote procedures concurrently (see Section 2.2.3).

As we have mentioned in Chapter 1, in NCS a client program usually has no a priori knowledge about the location of the remote procedure servers, and the only way to get this information is to access the location brokers during program execution. Also, the location of a remote procedure server may be changed by some application programs during the execution. So, without loss of generality, we will assume that no locations of remote procedure servers are known by a client program before its execution. In that case, all remote procedure calls within a client program are channeled through
location brokers. The information flow associated with the Apollo’s RPC system can be indicated in Figure 4.1.

![Figure 4.1. The relationship among client program, location broker, and servers]

The client program interrogates the location broker for the location of the server which exports a particular remote procedure. The RPC call is then placed on a queue associated with the server. After the completion of the call, it returns to the client program.

One of the important performance metrics for a concurrent program is the execution time. For a RPC program (which, as mentioned in Chapter 1, consists of client and server programs), a user usually sees the program’s execution through its client part. And the execution time of a RPC program, from the user’s point of view, is the execution time of the client program. The execution of the server’s remote procedures is hidden from the user and the time of the execution is reflected in the RPC calls within the client program. So, in the rest of this chapter, we use the term "RPC program" to mean the client part.

Many of the articles which address the execution time estimation of concurrent programs in this sense are based on queueing theory [Lavenberg83]. Heidelberg and Trivedi [Heidelberg83] discuss analytic queueing models for programs with internal concurrency. Thomasian and Bay [Thomasian83] present several queueing network
models which may be used to analyse parallel processing of task systems. The queueing
theory is a powerful tool for the analysis of concurrent models, 
queueing network models with explicit solutions are not directly applicable to these 
systems [Chandy83] because of the internal program concurrency. Also, because of the 
huge number of states, models with a non-explicit solution are often not feasible. 
Further, the queueing models obtain the execution time from the system’s viewpoint by 
estimating all possible jobs. On the other hand, a user is often interested in the execu­
tion time of his own job, that is, is interested from the user’s viewpoint ([Zhou89], 
[Qin89]). This is particularly true in RPC programming. Based on the user’s 
viewpoint, we present in this chapter an execution time evaluation model which has 
explicit solution for simple RPC programs and a nondeterministic algorithm as well as 
upper and lower bounds for complex RPC programs.

4.2. The Evaluation Model

4.2.1. Syntactic Issues

From the user’s viewpoint, a RPC program is a program which executes on a local 
host and which calls, either sequentially and concurrently, remote procedures located 
on other hosts.

A RPC program executing on the local host may do several things. Firstly, it may 
execute program segments which are completely located on the local host; secondly, it 
may call several remote procedures in sequence; thirdly, it may call several remote pro­
cedures in parallel. Generally speaking, the execution time of a remote procedure call is 
much longer than the execution time of a local procedure call because the remote pro­
procedure call will involve overheads such as finding the location of the required server, marshaling the arguments and results, transferring messages over the network in both directions, and executing the remote procedure. From the descriptions of many RPC pioneers, as well as from our own experience, the execution time of a RPC call is 100 to 1000 times longer than a local procedure call of the same function ([Nelson81], [Birrell84], [Coulouris88], [ZhouRep88]). To analyse a RPC program (i.e., to analyse the elapsed time that a client sees), we will omit the program segments executed on the local host except for the segments which may be related to the concurrent control of the RPCs. That is, we idealise RPC programs to contain only RPCs and some necessary control segments, and consider all other local segments of the program as taking zero time to execute. This is motivated by the fact that we want to study programs which are dominated by the time spent in RPCs.

A sequential RPC program block (sequential block) is indicated by

\begin{verbatim}
BEGIN
  a; b; c
END
\end{verbatim} \hfill (4.2.1)

where \(a, b,\) and \(c\) are called atomic remote procedures (or simply, atoms). That is, no remote procedures are called again from these procedures. Sequential RPC program blocks offer no speedup in a distributed system, because of the remote procedure call semantics. The execution time of a sequential program block is the sum of the execution times of all its atoms. We use \(SEQ (a, b, c)\) to denote (4.2.1).

Our model allows remote procedure calls to be made concurrently. That is, we define a parallel block as

\begin{verbatim}
COBEGIN
  a; b; c
COEND
\end{verbatim} \hfill (4.2.2)
Chapter 4. Execution Time Evaluation

If the atomic procedures \( a, b, \) and \( c \) are supported on different server hosts then they can be executed in parallel and the execution time of the parallel block is simply that of the largest of the component execution times. But usually these remote procedures are allocated by the location broker to a set of available hosts. This means the evaluation of execution time will not be as so simple. We use \( PAR( a, b, c ) \) to denote (4.2.2).

Our model is concerned with programs constructed from a set of atomic remote procedures by the repeated application of the above block operators. The abstract syntax of these programs is quite simple:

\[
\text{prog} ::= \text{seq	extunderscore block} | \text{par	extunderscore block} | \text{atom}
\]

\[
\text{seq	extunderscore block} ::= \text{BEGIN} \{ \text{prog} \} \text{END}
\]

\[
\text{par	extunderscore block} ::= \text{COBEGIN} \{ \text{prog} \} \text{COEND}
\]

Fragment (4.2.4) is an example of RPC program (where \( A_i \) and \( A_{ij} \) are atoms). Its motivation is given in section 4.5.

\[
\begin{align*}
\text{BEGIN} \\
A_1; \\
\text{COBEGIN} \\
\text{BEGIN} \\
A_2; \\
\text{COBEGIN} \\
A_{21}; A_{22} \\
\text{COEND} \\
\text{END}; \\
A_3; \\
\text{BEGIN} \\
A_4; \\
\text{COBEGIN} \\
A_{41}; A_{42}; A_{43}; A_{44} \\
\text{COEND} \\
\text{END} \\
\text{COEND} \\
A_5 \\
\text{END}
\end{align*}
\]

If a block \( A \) is inside another block \( B \), we call \( A \) a member of \( B \). In the above, atoms \( A_{21} \) and \( A_{22} \) are (direct) members of a parallel block \( X \):
Chapter 4. Execution Time Evaluation

COBEGIN
  \( A_{21}; A_{22} \)
COEND

The atom \( A_2 \) and the parallel block \( X \) are again the (direct) members of the sequential block \( Y \):

BEGIN
  \( A_2; \)
  COBEGIN
    \( A_{21}; A_{22} \)
  COEND
END;

So, \( A_{21} \) and \( A_{22} \) are called the non-direct members of \( Y \) if we want to distinguish them from direct members.

If all direct members of a (sequential or parallel) block are atoms, we call this block a simple block. So, \( X \) is a simple parallel block whereas \( Y \) is not a simple block. Simple blocks are easier to evaluate.

Equivalently we can associate each program with a flowgraph [Marcotty86] indicated by Figure 4.2 below:

![Figure 4.2. Program flowgraph](image)

where \( s \) and \( f \) denote start and finish nodes, respectively. The flowgraph of a program is built up from its atomic remote procedure calls, denoted by Figure 4.3:
Figure 4.3. Flowgraphs of atoms

The flowgraph of the above two block constructors can be shown as in Figure 4.4. Here Figure 4.4(a) is the flowgraph of (4.2.1) and Figure 4.4(b) corresponds to (4.2.2), respectively.

Figure 4.4. The flowgraphs of the three block constructors

By repeated application of the sequential and parallel constructors, one can associate with each RPC program an equivalent program flowgraph. The flowgraph of the previous RPC program (4.2.4) is given in Figure 4.11.

Assertion 4.1. The flowgraph of a RPC program is acyclic.

Proof: If there is a cycle, there must be a block which is a member of itself. This is impossible because each time one applies a block operation to one or several
blocks, those blocks become the direct member of a higher level block. That is, a block cannot have itself as one of its members.

### 4.2.2. Semantic Issues

Now we shall formalise the situation indicated in Figure 4.1. If we denote $P$ as the set of remote procedure atoms and $N$ as the set of server hosts, then we can represent the location broker by a function $m : P \rightarrow N$. Further, let us model the execution time of a remote procedure by a function $t : P \rightarrow \text{reals}$. Given a remote procedure $p : P$, the location broker tells us the host $m(p)$, and the execution of $p$ will take time $t(p)$.

The operational semantics of Figure 4.1 shall be made precise by describing an algorithm which determines the execution time of a RPC program. This algorithm consists of a "colouring game" on the associated flowgraph $g$. We use the following colour scheme for flowgraph arcs:

<table>
<thead>
<tr>
<th>Colour</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td>white</td>
<td>initial condition</td>
</tr>
<tr>
<td>blue</td>
<td>in RPC queue</td>
</tr>
<tr>
<td>red</td>
<td>executing</td>
</tr>
<tr>
<td>black</td>
<td>completed</td>
</tr>
</tbody>
</table>

and all arcs progress through the colours in this order. The game has the invariant:

**All red arcs are associated with a different server.**

Further, we say that a blue arc can execute if no red arc is associated with the same server.

The game involves assigning a value to an attribute $\text{CompletionTime}$ for each node and each arc of the flowgraph. It also involves a single global variable, denoted $\text{CurrentTime}$ (with initial value 0).
**Algorithm 4.1.**

Initialisation: Assign $CompletionTime = 0$ at start node; Colour blue all arcs leaving start node;

*While* there exist blue arcs which can execute {
  select a blue arc $v$ which can execute;
  colour $v$ red;
  $v.CompletionTime = CurrentTime + v.ExecutionTime$;
}

*While* red arcs remain {
  select red arc $x$ with minimum $x.CompletionTime$;
  colour $x$ black;
  $CurrentTime = x.CompletionTime$;

  denote target node of $x$ by $n$;
  *If* all arcs $y$ entering $n$ are black {
    $n.CompletionTime = \max(y.CompletionTime)$;
    colour blue all the arcs leaving $n$;
  }

  *While* there exist blue arcs which can execute {
    select a blue arc $v$ which can execute;
    colour $v$ red;
    $v.CompletionTime = CurrentTime + v.ExecutionTime$;
  }
}

Program execution time = the $CompletionTime$ of finish node;

There are two sources of nondeterminism in the algorithm, corresponding to the selection operations. The second of these is the most significant, corresponding to the scheduling of the queues in Figure 4.1. That the execution time depends on the selection strategy is easy to see. A simple example is provided by the following flowgraph (Figure 4.5):
where $a$, $b$, and $c$ are atomic remote procedures with unit execution time. Consider $a$ and $b$ to be mapped to host 1 and $c$ to be mapped to host 2. If procedure $a$ is selected first the overall execution time is 3 units, while if procedure $b$ is selected first the overall execution time is 2 units ($a$ and $c$ can execute in parallel).

4.2.3. Performance Measures

Given a location broker map $m$, a RPC program $g$ will execute (according to our model) in a time denoted $T_m(g)$. This time depends on the topology of the flowgraph of $g$, and the scheduling of the atoms over hosts. Our problem of interest is that repeated executions may take place, with a different location broker map each time. If we denote by $M$ the set of maps of interest, then the standard metrics are

$$T_{\min}(g) = \min_{m:M} T_m(g); \quad T_{\max}(g) = \max_{m:M} T_m(g);$$

$$T_{av}(g) = \exp(T_m(g)) = \sum_{m:M} T_m(g) pr(m). \quad (4.2.5)$$

where $pr$ is a probability function defined over the set of maps $M$. We are only interested in the equiprobable case, namely,

$$pr(m) = \frac{1}{S}; \quad \text{where } S = \text{card}(M).$$
Chapter 4. Execution Time Evaluation

It is easy to estimate the execution time of a sequential block by simply adding the execution times of all its direct members. That is, let $S$ be a sequential block defined as

$$S = SEQ(A_1, A_2, ..., A_n),$$

where $A_i$'s ($i = 1, 2, ..., n$) are $S$'s direct member blocks. Let $T_i$ ($i = 1, 2, ..., n$) be the estimated execution times of those members, then the execution time of $S$ is

$$T_S = \sum_{i=1}^{n} T_i. \quad (4.2.6)$$

But for general situations, it is very difficult to obtain the explicit solution. The next section gives the solution for a special case.

4.3. Simple Parallel Block Special Case

4.3.1. Analysis

In this section we consider the special case of programs involving the parallel execution of atomic procedure calls. These programs have the structure indicated in the equation

$$PAR(p_1, p_2, \ldots) \quad (4.3.1)$$

and equivalently have a flowgraph of the following form (where $p_i$ are atoms).

![Flowgraph of the simple parallel block](image-url)
Chapter 4. Execution Time Evaluation

Now given \( n : N \), we denote by \( m^{-1}(n) \) the set of remote procedures that execute on the host \( n \). If we denote by \( P \) the set of all remote procedures involved in the (4.3.1), then \( A_n = P \cap m^{-1}(n) \) is the subset of \( P \) that will execute on \( n \). In Figure 4.1 these are the remote procedures that will be placed on the queue of the server \( n \). To get a explicit solution we need to make a further assumption, namely, that all procedures execute in a standard time.

\[
t(p) = 1 \quad \text{for all } p : P
\]  
(4.3.2)

The analysis is now simple. The time needed to execute the queue on node \( n \) is the sum of the queue components

\[
T_m(A_n) = \sum_{p : A_n} t(p). 
\]  
(4.3.3)

while the time needed to execute all queues is the time of the largest queue.

\[
T_m(g) = \max_{n : N} T_m(A_n). 
\]  
(4.3.4)

Because of (4.3.2), (4.3.3) can be written as

\[
T_m(A_n) = \text{card}(A_n). 
\]  
(4.3.5)

4.3.2. Explicit Solution

We characterise the set \( M \) of maps as follows. Each \( m : M \) is a map from the finite set \( P \) to the finite set \( N \) where we denote

\[
L = \text{card}(N); \quad k = \text{card}(P). 
\]  
(4.3.6)

When maps are being set up in the location broker the procedures themselves are not distinguished. It is their allocation to a host that is of interest. Hence we identify a distinct map with the allocation of \( k \) identical "balls" to \( L \) different "boxes". This is a standard problem in combinatorics and its solution is well-known ([Bogart83], [Ke81]).

The maximum and minimum measures are easy. We have:

Assertion 4.2. If \( g \) is a simple parallel block, then
Chapter 4. Execution Time Evaluation

\[ T_{\text{max}}(g) = k ; \quad T_{\text{min}}(g) = \left\lfloor \frac{k}{L} \right\rfloor. \] (4.3.7)

**Proof:** The first formula corresponds to the case where all procedures are mapped to a single host and is evidently the worst case. The second corresponds to the case where the procedures are distributed as evenly as possible, and is the best case. Here \( \left\lfloor x \right\rfloor \) denotes the integer least upper bound.

Now we derive the average measure. First we compute \( S \), the total number of maps. From combinatorics results ([Bogart83], [Ke81]) we know this is a type-2 distribution problem and that

\[ S = d_k^{(2)}(L, [0, \infty)) = \left\lceil \frac{L+k-1}{L-1} \right\rceil \] (4.3.8)

where \([0, \infty)\) is the restrict condition of balls in a single box. But from the problem we know the maximum balls within a single box can not be more than \( k \). So here the restrict condition can be changed to \([0, k]\).

If we denote by \( Q_{i}^{k/L} \) the number of maps with \( \max_{n \in N} |A_n| = i \), then

\[ T_{av}(a) = \frac{1}{S} \left( 1 \times Q_1^{k/L} + 2 \times Q_2^{k/L} + \cdots + k \times Q_k^{k/L} \right) \] (4.3.9)

\[ = \frac{1}{S} \sum_{i=1}^{k} i \times Q_i^{k/L} \]

The combinatorial meaning of \( Q_i^{k/L} \) is: \( k \) identical balls are to be distributed to \( L \) different boxes and the maximum number of balls within a single box is exactly equal to \( i \) and at least one box has this number of balls. If the maximum number of balls within a single box is less than or equal to \( i \), then [Ke81]

\[ d_k^{(2)}(L, [0,i]) = \sum_{0 \leq j \leq L} (-1)^j \left\lceil \frac{L+k-(i+1)j-1}{L-1} \right\rceil \] (4.3.10)

So,

\[ Q_i^{k/L} = d_k^{(2)}(L, [0,i]) - d_k^{(2)}(L, [0,i-1]), \] (4.3.11)

\[ i = 1, 2, \cdots, k, \text{ and } d_0^{(2)}(L, [0,0]) = 0. \]

It is evident that \( S = \sum_{i=1}^{k} Q_i^{k/L} \).
Chapter 4. Execution Time Evaluation

So we have the following:

Assertion 4.3. If \( g \) is a simple parallel block, then the average execution time of \( g \) is

\[
T_{av}(g) = \frac{1}{S} \sum_{i=1}^{i=k} i \times Q_i^{k/L} .
\]

where \( S \) and \( Q_i^{k/L} \) are defined in (4.3.8) and (4.3.11), respectively.

4.3.3. Comparison Between Calculated and Simulated Results

It is interesting to know how the results calculated from the above formulas compare with the results obtained from a simulation of Algorithm 4.1. The next two graphs show that these two kinds of results are reasonably close to each other.

In Figure 4.7(a), we set the number of RPCs (\( k \)) in a simple parallel block to 20 and let the number of hosts (\( L \)) in the system change from 1 to 20. Then the calculated and simulated execution times are compared. The calculation result in this case is shown in solid line and the simulation result in dashed line. It is seen that when \( L = 1 \), both times are equal to 20. When \( L \) increases, both times decrease. The decrement is very large before \( k = 5 \) and relatively large between \( 5 < k < 10 \). After \( k = 10 \), the execution times decrease very slowly. The above analysis correspond to our intuition.

In Figure 4.7(b), we set the number of hosts (\( L \)) of the system to 10 and let the number of RPCs (\( k \)) in a simple parallel block change from 1 to 40. Then the calculated and simulated execution times are compared. The calculated result (solid line) in that case is a linear function of the RPC number in the parallel block and the simulated result (dashed line) coincides with the calculated result.

During our simulation, we set the execution time of each RPC call to 1 time unit, and simulated each (\( k, L \)) pair 1000 times. Each simulation time corresponds to a location broker mapping \( m \) which randomly maps remote procedures to server hosts. The
Chapter 4. Execution Time Evaluation

The average value of these 1000 simulations is the execution time when there are \( k \) RPCs and \( L \) hosts in the system.

![Figure 4.7](image)

**Figure 4.7.** Average execution time and simulation results

(a) \( L \) changes (\( k = 20 \)).  (b) \( k \) changes (\( L = 10 \)).

(Solid line: Calculation; Dashed line: Simulation)
4.4. Estimation of Lower and Upper Bounds

To this point we have established the explicit solution for sequential blocks and simple parallel blocks. Although the explicit solution for a general structured distributed program cannot be obtained at this moment, we can estimate the lower and upper bounds of execution time of such a program by simplifying the program structure to feasible forms. These are the tasks of the following two sections.

4.4.1. Lower Bound

We define an extended parallel block as

\[
\text{COBEGIN} \quad B_1 ; B_2 ; \cdots ; B_k \quad \text{COEND}
\]

(4.4.1)

where \(B_i\) is a simple parallel block or an extended parallel block. This is a recursive definition.

We now simplify the execution time estimation of an extended parallel block to a feasible form. If there are \(L\) hosts, we define the lower bound of the average execution time of an extended parallel block \(A\) as in formula (4.4.2).

\[
TL_{av}(A) = \max \left( \frac{\sum_{i=1}^{k} TL_{av}(B_i)}{L} , \max_{i : \{1..k\}} (TL_{av}(B_i)) \right)
\]

(4.4.2)

If \(B_i\) is a simple parallel block, then \(TL_{av}(B_i)\) is calculated by using the formulas in section 4.3. That is, in that case we have \(TL_{av}(B_i) = T_{av}(B_i)\). Otherwise, \(TL_{av}(B_i)\) is calculated by using (4.4.2) again. That is, the calculation is also recursive. So formula (4.4.2) is feasible.

By extending the (4.4.2) definition to a RPC program \(g\), we can have the lower
bound execution time of $g$. The calculation can be carried out from the inner simple blocks of $g$ to the outer blocks. For parallel block, formula (4.4.2) is applied, while for sequential blocks, formula (4.2.6) of section 4.2.3 is applied. The claim that $TL$ is a lower bound is justified by the following result.

**Assertion 4.4.** If $g$ is a RPC program, then

$$T_{\text{min}}(g) \leq TL_{av}(g) \leq T_{av}(g).$$

**Proof:** The left part of (4.4.3) is evident because $TL_{av}(g)$ considers some sequential allocation of atoms in parallel blocks of program $g$ while $T_{\text{min}}(g)$ only considers the even allocation of atoms. The right part of (4.4.3) is also true because of the following claim. (1) In (4.4.1), if $k = 1$, then $TL_{av}(A) = T_{av}(A)$. (2) If $k > 1$ and $B_i$ and $B_j$ are two parallel blocks, then the calculation of (4.4.2) considers them independently, while in fact they may execute concurrently. In that case we have $TL_{av}(A) \leq T_{av}(A)$. (3) The meaning of (4.4.2) is to consider $k$ sub-extended parallel blocks as being allocated on $L$ hosts as evenly as possible, while it is not always the case. So we still have $TL_{av}(A) \leq T_{av}(A)$. This proves that (4.4.3) holds.

### 4.4.2. Upper Bound

If $g$ is a RPC program and $G$ is its flowgraph, we define the *level* of an atom $p : g$ as the length of the *longest* path from the start node $s$ to the end node of $p$ and denote it as $p.level$. Now the upper bound of $g$’s average execution time $TU_{av}(g)$ can be defined as follows:

1. **Construct simple parallel block $A_i$** such that if $p : A_i$, then $p.level = i$, $i = 1, \ldots, M$ and $M$ is the maximum level of $G$;
2. **Calculate $T_{av}(A_i)$** by using formulas in section 4.3;
3. **$TU_{av}(g) = \sum_{i=1}^{M} T_{av}(A_i)$**.
Chapter 4. Execution Time Evaluation

It is easy to see that the calculation of the upper bound is feasible. Again, the claim that $TU$ is indeed an upper bound is justified by the following result.

**Assertion 4.5.** If $g$ is a RPC program, then

$$T_{av}(g) \leq TU_{av}(g) \leq T_{\text{max}}(g) \quad (4.4.4)$$

**Proof:** The right part of (4.4.4) is evident because $T_{\text{max}}(g)$ corresponds to all atoms as sequentially executed, while $TU_{av}(g)$ considers some degree of parallelism. The left part is also true because atoms in different levels may execute concurrently, while $TU_{av}(g)$ views them as strictly sequential execution. So (4.4.4) holds.

### 4.4.3. Simulation Results

It is very interesting to know how the execution time changes within the lower and upper bounds in the general case. We will show this by using a complex RPC program. The flowgraph of the program is in Figure 4.8. It has 54 RPCs and 11 nodes (including start and end nodes).

Our purpose here is to show the variation of the program’s execution time as well as lower and upper bounds when the number of hosts in the system changes. It is infeasible to obtain a explicit solution for the program. So, we use Algorithm 4.1 to simulate the execution time of the program. Then the appropriate lower and upper bounds are calculated. Figure 4.9 shows the result where upper bound is in solid line, lower bound is in dashed line, and simulation result is in dotted line. It is seen that the execution time and lower/upper bounds decrease dramatically as host number increases from 1 to 10. After that, the decrement of simulated execution time and lower/upper bounds is very slow. The simulated execution time keeps within the lower and upper boundary as $L$ changes. This is consistent with Assertion 4.4 and Assertion 4.5.
Chapter 4. Execution Time Evaluation

Figure 4.8. Flowgraph of a complex RPC program

Figure 4.9. Simulated execution time and lower and upper bounds when $L$ changes ($k = 54$ and the fixed flowgraph of Figure 4.8).
(Solid line: Upper bound; Dashed line: Lower bound; Dotted line: Simulation)
4.5. Applications of the Model

4.5.1. Example: A Seller-Buyer System

Suppose there is one seller and several buyers. The seller at first sends a message to each buyer by using a remote procedure provided by the buyer, describing the nominal price, amount and performance of some goods. After a buyer receives the message, he will decide whether he is interested in the goods and if so will submit a bid involving an amount and price. When the seller receives the bids from the return values of the remote procedures, he will commit to a contact with the buyer with the best offer, again using a remote procedure provided by the buyer. The seller resides on the local host, while the buyers are allocated on the distributed system (which consists of 5 hosts). The structure of the RPC program \( g \) corresponds to the flowgraph of Figure 4.10, where \( A_2 \) to \( A_6 \) are remote procedures provided by buyer 1 to 5 which can be used by the seller to send out the message and get the return bids, and \( A_7 \) is the remote procedure call the seller sends out to the best buyer. Remote procedure call \( A_1 \) is used by the seller to do some preparation before issuing the message.

Figure 4.10. Flowgraph of the simple example

Suppose all of the \( A_i \)'s (i=1 to 7) are atomic remote procedure calls and the execu-
tion times for $A_1$ to $A_7$ are all 1. For convenience, we denote the parallel block of the program as C. We have

$$T_{av}(C) = \frac{1}{S} \sum_{i=1}^{i=5} i \times Q_i^{5/5}$$

$$= \frac{1}{126} \times (1\times1 + 2\times50 + 3\times50 + 4\times20 + 5\times5) = \frac{356}{126} \approx 2.8.$$  

That is, the execution time for C is 2.8 time units. So the evaluation time of this RPC program is $T_{av}(g) = 1 + 2.8 + 1 = 4.8$ time units.

Now, if we assume the distributed system has only 3 hosts instead of 5, then

$$T_{av}(C) = \frac{1}{S} \sum_{i=1}^{i=5} i \times Q_i^{5/3}$$

$$= \frac{1}{21} \times (1\times0 + 2\times3 + 3\times9 + 4\times6 + 5\times3) = \frac{72}{21} \approx 3.4.$$ 

So the estimated execution time of this RPC program will be 5.4 time units. In each case the lower bound and upper bound of the average execution time is the same as the average execution time.

**4.5.2. Example: An Extended Seller-Buyer System**

Next we extend the above seller-buyer system by adding two wholesalers. That is, the seller concurrently sends out the goods information to some buyers and two wholesalers. These two then send the information to their own customers (buyers) with their own comments which may influence these buyers. All the bids return to the seller. Finally, the seller will send the goods to the buyer with the best offer. The structure of RPC program $g$ corresponds to the flowgraph of Figure 4.11(a). Here $A_1$ and $A_5$ are the same as the $A_1$ and $A_7$ of the last example, respectively. $A_2$ is the remote procedure of the first wholesaler, while $A_{21}$ and $A_{22}$ are its customers’ remote procedure calls. $A_4$ is the RPC of another wholesaler and $A_{4i}$ are the RPCs of the customers. $A_3$ is the seller’s customer. Here $A_i$ and $A_{ij}$ are atoms and their completion time are all 1 unit.
This time the derivation of an explicit solution to $T_{av}(g)$ for this program is intractable. Instead we shall indicate the use of lower and upper bounds to estimate the average execution time of $g$.

Figure 4.11(b) is the restructuring of Figure 4.11(a) using extended parallel blocks, and Figure 4.11(c) is the restructuring of Figure 4.11(c) using levels. If we assume that there are 5 hosts, then we have

$$T_{Lav}(g) = 1 + \max\left(\frac{(1+1.3)+1+(1+2.4)}{5}, 3.4\right) + 1 = 5.4,$$

$$T_{Uav}(g) = 1 + 1.9 + 3.3 + 1 = 7.2.$$ 

So, $5.4 \leq T_{av}(g) \leq 7.2$. If there are 3 hosts, then we have $5.8 \leq T_{av}(g) \leq 8.2$.

Figure 4.11. Structure of the complex example

4.6. Remarks

A model for the evaluation of the execution time of a distributed program is developed in this chapter. We evaluate this execution time from the user’s viewpoint.
Chapter 4. Execution Time Evaluation

instead of from the system's viewpoint, because a programmer is usually interested in his own program instead of the whole system.

An explicit solution for general distributed programs is not available at this time. We provide a nondeterministic algorithm to evaluate the estimated executing time. This algorithm characterises a single execution for a single location broker map which maps all remote procedures of a distributed program into the given distributed system. As we are interested in the average behaviour over the set of all such maps, we then provide the explicit solution for sequential blocks and simple parallel blocks. After that the upper and lower bounds estimated execution time for general-structured distributed programs are developed. The correctness of the explicit solution as well as upper and lower bounds are supported by the simulation results.
Chapter 5
RPC Transaction Management

5.1. Background and Related Works

Transaction management is a well established concept in database systems. Two kinds of transactions are defined over an object system (a system with a collection of objects, where an object can be a database file or an entry of the database file): the atomic transaction and the non-atomic transaction. An atomic transaction is defined as a sequence of operations which has the following two properties ([Cary81], [Liskov83], [Coulouris88], [ZhouRep90a]):

(1). Recoverability. The overall effect of a transaction is all-or-nothing: either all of the objects addressed by the transaction remain in their initial states before the transaction or are all changed to their final states.

(2). Indivisibility. The partial effects of a transaction are invisible to other transactions. That is, if the objects being modified by a transaction are observed over time by a second transaction, then this second transaction will either always observe the initial states before the first transaction, or the final states after the first transaction.
A non-atomic transaction is similar to an atomic transaction except that the second property no longer necessarily holds. That is, the partial effect of a non-atomic transaction may be accessed by other non-atomic transactions. The implementation of non-atomic transaction is easier and the execution overhead is smaller compared with atomic transaction management. Non-atomic transactions are needed when the duration of a transaction is long so that it is intolerable to wait until a transaction commits before the next transaction begins to process the same data objects [Korth88].

The notion of combining the transaction concept with the RPC facility is not new but the combination has not been studied to any extent. Existing RPC implementation or proposals usually do not deal with the transaction aspect of a RPC call or a set of RPC calls. The reasons can be quoted from Nelson [Nelson81]:

First, the policy and expense of making each remote call atomic is too great a burden for many clients. Second, remote procedures and atomicity are basically independent notions that require more investigation and experience before being tied together.

Almost all the existing RPC implementations use at-most-once [Coulouris88] as the calling semantics. That is, when a return message is received without an error flag, the user can be confident that the remote procedure has been executed exactly once. On the other hand, if a return is received with an error flag, the user does not know if the RPC has been executed or not. For example, if the RPC request message is lost, the remote procedure has not been executed; while if the reply message from the remote procedure is lost, then the remote procedure has been executed. In both cases the client knows from the exception that something is wrong between the client and server. The client cannot tell if the remote procedure has been executed or not.

The motivation for transaction-oriented RPC calls are obvious. For example, in our distributed calendar application, a user usually issues a meeting which involves a
group of people. Suppose everybody in the group is crucial to the meeting. That is, if anybody cannot attend the meeting, then the meeting period must be re-arranged by the issuer. As we know, the calendar database entries of the participants may be located across several different hosts. So, the meeting arrangement involves a collection of RPC calls, one for each participant. Now if any one of these calls goes wrong (such as the relevant hosts or links are down, or the time periods of some participants are already occupied and cannot be re-allocated) during the meeting arrangement call, we need a method to roll back those RPC calls that have been completed.

We take the point of view that the transaction facility can be embedded into the system instead of into the clients’ programs. In that case, the clients’ burden can be reduced. For instance, in the above distributed calendar application, a RPC transaction call can be provided for users that require their calls be recoverable and/or indivisible. Because a transaction-oriented RPC call is more expensive than a non-transaction-oriented RPC call, both kinds of calls should be provided. Also, we think the techniques and experiences of both transaction management (mostly from database systems) and RPC are mature. The integration of RPC and transaction management is now feasible.

Liskov and her colleagues considered the atomicity of RPC from the viewpoint of programming languages. Concepts such as guardians, actions [Liskov83], atomic data types [Weihl85], and promises [Liskov88] are introduced to ensure the atomicity of some segments of a program. But a programmer has to incorporate into his program many "new" segments that deal with the atomicity of the RPC calls. In such a case the programming is not so transparent. They also do not consider the atomic transactions with client/server structure in usual RPC-oriented systems and applications.

We are not going to consider the full atomic RPC transaction within this dissertation. Instead, we only consider non-atomic transactions. So, from now on when we say transaction, we understand it to mean non-atomic transaction.
Two problems are dealt with in this chapter:

- **Single RPC transaction management**: The transaction management of a single RPC call.
- **Parallel RPC transaction management**: The transaction management of a set of parallel RPC calls.

Both problems are considered within the client/server paradigm and are transparent to users.

The remainder of this chapter is organised as follows. Section 5.2 introduces the RPC transaction models used in our transaction management. Section 5.3 presents the design of a management system for single RPC transactions, and describes the properties of the system. Based on that, Section 5.4 describes a system for managing a set of (parallel) RPC calls. Properties of the extended system are also described.

### 5.2. RPC Transaction Models

#### 5.2.1. The RPC Model

In this section we consider the RPC model from the viewpoint of a user process.

Let

\[ P = \{ p | p \text{ is a remote procedure call} \} \] and \[ D = \{ d | d \text{ is a data item of the object system} \}. \]

Please note that two \( p \)'s are treated differently in \( P \) even if they use the same program segment to make their calls.

After we make a remote procedure call, the call may return successfully or may fail. There may be several reasons for the failure of a call. For example, in Listing 2.5 of Chapter 2, there is a function called `ca_find_a_cal_func()` which sets an
exception segment before the real RPC call as follows:

```c
if (fst.all != pfm_$cleanup_set) {
    if (fst.all != status_$ok)
        fprintf(stderr, "Exception raised in find_a_cal -
        %s
", error_text(fst));

    switch (fst.all) {
        case rpc_$comm_failure:
            fprintf(stderr, "Communication error\n");
            break;
        case rpc_$wrong_boot_time:
            fprintf(stderr, "Server fails\n");
            break;
        default:
            fprintf(stderr, "Unknown error\n");
            break;
    }
    rpc_$free_handle(rh, &st);
    return -1;
}
```

That is, it catches several errors such as "communication error" and "server error." There are also other errors it does not catch but which may happen during a RPC call.

We can divide all these errors into two classes. The first class includes those errors for which we are definitely sure that the call has not been performed, while the second class includes those errors for which we cannot tell whether the call has been performed or not. For example, by having a "server error", we are sure that the RPC has not been performed. But if we have a "communication error," the client will not able to tell if the call has been performed or not because we do not know the error happened before or after the calling request arrived at the destination host.

We define the effect of a RPC call as the processing of a data item of the object system. Hence we can abstract the type of a RPC call as a mapping

\[ c : P \times D \rightarrow \{ OK, FL, US \}. \]

**OK** This means that no error occurred during the RPC's execution. By \( c(p, d) = OK \), where \( p : P \) is a RPC call and \( d : D \) is a data item of the system, we mean that the RPC call was successful (i.e., the remote procedure performed its assigned task).
Chapter 5. RPC Transaction Management

FL This means accessing failure. By $c(p, d) = FL$, we mean that the destination server (the server that exports the remote procedure $p$) did not perform its assigned task. This occurred, for example, because the arguments between the client and server did not match, the versions were different, or because the object managed by the server is not accessible to the client. This means that the RPC is known not to have executed.

US This means unknown state. By $c(p, d) = US$, we mean that the client cannot tell if the RPC completed or not. For example, the destination host (the host on which the server exporting procedure $p$ resides) or the server may have failed, or the links between the client and server may have failed (that is, the client host and the server host belong to two partitioned sub-networks). In these cases the RPC request itself may be lost, or the return message may be lost. Hence we do not know if the remote procedure has been executed or not.

We may have several strategies to deal with the case of receiving a US value. For example, a strategy might be:

- Query the system to determine if the RPC really happened.
- If the RPC occurred then provide the return values. Otherwise explicitly call the RPC again.

This strategy is not useful, however, because the query may last an intolerably long time. For example, if the links between the server and the client are down, the query will never return a correct result until the link is recovered. Hence we adopt the following strategy:

- We provide a rollback operation that will reverse the call if it happened and do nothing if it did not happen. The operation is defined as:

  $$ r : P \times D \rightarrow \{OK, FL, US\}. $$

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

The rollback operation is managed by the system. When a rollback operation is issued, the user can go back to his own work immediately instead of waiting for the result. The system will guarantee that the RPC will be rolled back. In order to have confidence that a RPC is really rolled back, the system may keep using the \( r \) operation until an \( OK \) is returned.

Of course, some operations cannot be rolled back. For example, suppose a RPC call is to print a cheque. When the rollback request comes, if the printing task is still in the printer queue, it is easy to do the rollback; but if the printing task has been performed, then the computer system can do nothing about that. In our following discussion, we consider only RPC operations that can be rolled back.

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

Now consider that there is only one user process and a RPC call \( c(p) \) is to be used to change the state of a data item \( d \) from \( S_0 \) to \( S_1 \). If \( c(p) = OK \), then \( d \) is in \( S_1 \). If we want to roll it back, then \( r(p) = OK \) will guarantee that the RPC has been rolled back, that is, \( d \) is now in \( S_0 \). If \( c(p) = FL \), then \( d \) is still in \( S_0 \). If we also do the rollback operation, then \( r(p) = OK \) means that there is no effect caused by the operation. That
is, \( d \) remains in \( S_0 \). When \( c(p) = US \), the user does not know what is the current state of \( d \). If it is in \( S_0 \), then \( r(p) = OK \) has no effect on \( d \), and \( d \) still remains in \( S_0 \); if it is in \( S_1 \), then \( r(p) = OK \) will make \( d \) back to \( S_0 \). Figure 5.1 illustrates the above discussion.

![Diagram](image)

**Figure 5.1. RPC and its rollback operation**

There are other possibilities when \( c(p) = US \). Consider the case where the destination system (\( B \)) in Figure 5.2(a) crashes after it receives the RPC request but before it executes the request. Moreover, the RPC call is considered to resume after host \( B \) recovers. On the other hand, the calling process will receive a \( US \) message due to a time out event (any reply from the server after that will be ignored) and a rollback operation will be issued. Of course, the rollback operation cannot succeed before host \( B \) recovers. But after host \( B \) recovers, both the RPC and the rollback operations will be carried out. Now, if the rollback operation performs first and \( r(p) = OK \), the user may think that the RPC has been rolled back, and \( d \) is in \( S_0 \). But after the rollback operation (which leaves \( d \) in \( S_0 \)), the original RPC request performs and makes \( d \) change from \( S_0 \) to \( S_1 \)!

One possible solution to this problem is to have the recovery process of each host kill all incomplete RPC operations before it returns normal execution. In that case, the original RPC request in the above scenario will be killed during recovery and the user
can have confidence that $d$ will remain $S_0$ after the rollback operation. Another possible solution is to use the First-In-First-Served (FIFS) strategy to order all operations of a server (except parallel operations). In that case, the incomplete RPC operation will perform first and then the rollback operation will roll back $d$ to $S_0$. We adopt the latter strategy.

![Diagram](image)

**Figure 5.2. Some possibilities when return is US**

A second possibility is that the calling host ($A$) may crash just after its process issued the RPC call $c(p)$, as indicated in Figure 5.2(b). Then, after host $A$ recovers and the process restarts, it will do $c(p)$ again without knowing that there is already such a RPC call under execution on the destination host $B$. In this case, $c(p)$ will be performed twice. Nelson [Nelson81] discussed this problem and called the existing request as an orphan. An orphan (or an orphaned call), according to Nelson, is a remote request that has some ancestor caller executing on a crashed host. A possible solution of this problem is to ensure that one of $A$'s responsibilities during recovery is to have all its orphaned calls exterminated before it resumes normal operation. Nelson proposed several extermination algorithms which kill all orphaned calls. Basically, the extermination algorithm works as follows. Each host keeps two sets in stable storage [Lam- son81]: one contains the calling hosts (for example, host $A$ above) of all incoming calls,
and another contains the destination hosts (for example, host \( B \) above) of all outgoing calls. During crash recovery, these sets are used to exterminate orphaned calls and to notify calling hosts of the extermination.

Finally let us consider that there are several user processes in the system. Let process \( P_1 \) issue \( c(p) \) and process \( P_2 \) issue \( c(q) \) (they are in different RPC transactions). Also let \( c(p) \) change \( d \) from its current state to state \( S_1 \), and \( c(q) \) change \( d \) from its current state to state \( S_2 \). Now suppose \( d \) is in state \( S_0 \) and \( P_1 \) and \( P_2 \) perform their operations in the following order on destination host \( B \):

\[
P_1: c(p); \quad P_2: c(q); \quad P_1: r(p);
\]

Then \( P_1 \) will think \( d \) is in \( S_0 \) while \( P_2 \) will think \( d \) is in \( S_2 \)! Alternatively the rollback operation may not have the desired effect because its current state is \( S_2 \) instead of \( S_1 \).

There are several ways to solve this problem. One way is to let \( P_1 \) lock \( d \) until the transaction that \( P_1 \) belongs to finishes (commits or aborts). RPCs that belong to other transactions (such as \( P_2 \)) will wait (or simply return \( FL \)) until \( P_1 \) releases its lock on \( d \). A more sophisticated method uses the \textit{two-phase lock protocol} [Bernstein87] to manage all RPC calls. For simplicity, here we use the first solution. We do not formally define \textit{lock} and \textit{release} operations because we want to concentrate our attention on RPC transaction algorithms. Moreover, we let a RPC return \( FL \) if its data item is already locked. We assume that the servers have some facilities to manage the lock and release of the data items they manage (see Section 5.2.4). When we say a RPC locks a data item \( d \), we mean here that RPCs of other transactions cannot write to \( d \) until the RPC releases \( d \). Read operations on \( d \) are allowed. That is, some partial effect of a transaction is accessible (readable) by other transactions.
5.2.2. Single RPC Transaction Model

Now we can describe our single RPC transaction model. We define a single RPC transaction as \( T = \{ c(p, d) \} \) where \( c(p, d) \) is a RPC call. If there is no confusion, we simply denote \( T \) as \( T = \{ c(p) \} \). The semantics of a single RPC transaction is that after issuing the transaction, \( c(p) \) will be executed if no errors occur, or will be rolled back if something is wrong. We can picture \( T = \{ c(p) \} \) as Figure 5.3.

![Figure 5.3. Semantics of the single RPC transaction](image)

5.2.3. Parallel RPC Transaction Model

We define a parallel RPC transaction as \( T = \{ c(p_1, d_1), \ldots, c(p_m, d_m) \} \). Here \( c(p_i, d_i) (1 = 1, \ldots, m) \) is a RPC call, and \( p_i : P, d_i : D \). If there is no confusion, we also denote \( T \) as \( T = \{ c(p_1), \ldots, c(p_m) \} \). The semantics of a parallel RPC transaction is that after issuing the transaction, all components \( c(p_i) \) of \( T \) will be executed, or if any one of them fails, all the executed components will be rolled back. The execution of all \( c(p_i) \) in \( T \) is in parallel. Some parallel primitive, such as COBEGIN–COEND we described in Section 2.2.3, can be used for parallel execution of remote procedures.
Sequential executed transaction can be easily established from the parallel model. Figure 5.4 indicates the semantics of a parallel RPC transaction. In this figure $d_i$ (i=1, ..., $m$) are data items processed by the transaction.

![Figure 5.4. Semantics of the parallel RPC transaction](image)

5.2.4. Accessing Servers

As pointed out in Section 1.3.1, in the client/server model, server processes manage objects (data items) and client processes access these objects by using the remote procedures exported by servers. Each server can then be viewed as a monitor [Hoare74], and the objects it manages can be viewed as shared resources. The server enforces the following rules:

1. The shared objects of the server can be accessed only by the remote procedures provided by the server.

2. Two or more clients can access the server simultaneously, but only one client at a
time can succeed in changing the same object of the server. If a client is already using the object, other clients that want to change the object will have FL returns.

(3). The remote procedures may be called by any clients.

It may be argued that the second rule (2) is somewhat rigid — sometimes it is preferable to let a client call wait (for example, in a queue) a busy object (another client call is processing this object) instead of immediately return FL. But there are two problems in this approach. First, the client cannot predict how long he has to wait. That may create some deadlock conditions if a client processes more than one object. Second, some kind of queueing facility has to be built for each data object. If an object is simply an entry of a database file, then this may cause too much overhead to maintain the queues. Hence we let the client returns FL immediately if the accessing object is busy.

These rules guarantee mutual exclusion between a set of client processes that use the shared objects in a server. Usually, a server and all its objects are located at the same host. For example, in our distributed calendar system, the calendar database can only be accessed by using the server’s remote procedures. If two client programs want to change the same meeting entry, only one of them is allowed to do so at one time. Also, any client program of the server can call all remote procedures.

Another observation is that a remote procedure is far more complex than simple read or write operations used to model database-oriented transactions. Take our earlier flight reservation RPC as an example. The RPC may involve reading the reservation database, checking the required state and requirements, and writing back to the database if the customer’s requirements are satisfied. So, the database concurrency control techniques [Bernstein87] are not directly applicable to our model. Fortunately, constructing a server as a monitor can guarantee the server’s objects being accessed correctly, while the concurrency control techniques can contribute to this guarantee in the case of failures.
Chapter 5. RPC Transaction Management

The client/server relationship between two processes exists only for their duration of the interaction (client calls server and server returns the calling results). Thus a process that acts as a server at one interaction may become a client in another interaction. For example, if process $A$ calls a remote procedure of $B$, and $B$ calls a remote procedure of $C$. In that case, $B$ is a server from $A$’s viewpoint, and a client from $C$’s viewpoint. But we can divide $B$ into two parts, one acts as $A$’s server and another as $C$’s client. So, we still have logically client/server relationships.

5.3. Single RPC Transaction Management

5.3.1. The Manager

The structure of a RPC manager (RM) is indicated in Figure 5.5. It is used to manage single RPC transactions.

![Figure 5.5. Structure of the RPC Manager](image)

Each host in the network has a RM executing on it. Let each host be assigned a unique number in the range $1..N$. The set of RMs in the network can be expressed as...
Chapter 5. RPC Transaction Management

\{RM_1, RM_2, ..., RM_N\}. A RM consists of a managing server (called RM server) and three tables implemented in stable storage [Lampson81]. Information stored in these tables will not be affected by system failures. These tables are:

LET  \textit{Locally Executed RPC Table}. When a RPC is performed by a server in the host, it is reported to the RM server and stored in the LET. An entry in LET has two fields. If \( b_i : \text{LET} \), then \( b_i.rpc = a \), where \( a \) is the name of the completed RPC call. It is kept unique for each call (for example, a naming scheme similar to our event name definition of Section 3.2.1 can be used here. A \textit{rpc call name} can then be defined as \textit{ProcedureName.TimeStamp.SeqNo.HostID}. It is assigned by the host where the call is made and sent to the server host together with the RPC call). And \( b_i.host_{from} = h \), where \( h \) is the number of the client’s host. The function of LET table is to denote the executed RPCs and report them to the calling hosts. An entry of the table is defined as:

\begin{verbatim}
typedef struct let {
    /* name of the finished RPC call */
    char *rpc;
    /* client host number of the RPC call */
    int host_from;
} LET;
\end{verbatim}

UST  \textit{Unknown State Table}. When a RPC call provides the return US, then this call is recorded by the RM server into the UST. It is impossible at this point for the RM server to tell if the RPC has been performed or not by the destination host. There are three fields for each entry in UST. If \( b_i : \text{UST} \), then \( b_i.rpc = a \) (\( a \) is the unique name of the RPC call); \( b_i.host_{to} = h \) (\( h \) is the host number of the destination host); and \( b_i.time = t \) (\( t \) is the time the RPC call was issued and it is stamped by the caller’s host). Combining the local UST table with related LET entries of other RM servers, we can decide which entries in the UST have been executed and which have not. The definition of a UST entry is:
typedef struct ust {
    /* name of the unknown state RPC call */
    char *rpc;
    /* destination/server host number */
    int host_to;
    /* local Timestamp when call issued */
    long time;
} UST;

NRT

Needed Rollback Table. If the RM server decides that a RPC is to be rolled back, it is put into the NRT. Rollback commands for all entries in the table are issued periodically by the RM server. Each NRT entry has one field. If \( b_i : \) NRT, then \( b_i.rpc = a \) (\( a \) is the unique name of the RPC which is to be rolled back). A NRT entry is defined as:

```
typedef struct nrt {
    /* name of the needed rollback RPC call */
    char *rpc;
} NRT;
```

We denote the tables on host \( j \) as \( LET_j, UST_j, \) and \( NRT_j \), respectively.

5.3.2. Algorithms

Now we describe the algorithms used by each RM server in the network.

The RM server’s algorithm is described in Algorithm 5.1, where the LET, UST, and NRT tables and their counters are located in stable storage. When the RM server is invoked, all these tables are set to empty and their counters are set to 0 by the initialisation function. When a node recovers from a crash, these tables survive and can be used after the recovery. The constant \( \text{MAXENTRIES} \) is defined as the maximum number of entries in each of these tables. The RM server forks into five permanent concurrent processes after initialisation.
Algorithm 5.1.

```c
LET *let_table[MAXENTRIES]; /* LET table */
int let_ct;                 /* count of LET table */
LET *ust_table[MAXENTRIES]; /* UST table */
int ust_ct;                 /* count of UST table */
LET *nrt_table[MAXENTRIES]; /* NRT table */
int nrt_ct;                 /* count of NRT table */

Initialisation();

COBEGIN

/* periodically send out entries in LET */
send_my_let(let_table);

/* periodically rollback entries in NRT */
rollback_nrt(nrt_table);

/* listen to the sending of LETs by other
RM servers and process them */
listen_extern_lets(ust_table, nrt_table);

/* manage the RPC calls issued by its clients */
manage_rpc_calls(ust_table);

/* listen to local servers and put executed
RPC names into LET */
listen_local_servers(let_table);

COEND
```

The first function send_my_let() periodically groups all entries in the LET table according to their b.host_from fields, and sends them out to that host. If the sending of a group of entries (called a LET entry package) is successful, those entries are deleted. The algorithm of the function is as follows:
Chapter 5. RPC Transaction Management

Algorithm 5.2.

send my let (let_table)
LET *let_table[];
{
    int j, k;
    LET *grp_let[MAXENTRIES];

    while (TRUE) {
        j = the first host number (1);
        /* loop until all host numbers are checked */
        while (j != 0) {
            /* group all let_table entries which have the same
               b.host_from = j field into grp_let table */
            /* k is the returned number of entries in grp_let */
            k = group(grp_let, j, let_table);

            if (k != 0) {
                /* send out the LET entry package grp_let */
                /* if the sending is successful, delete
                   these entries */
                if (send_out(grp_let, k) == SUCCESS)
                    delete_let(grp_let, j, let_table);
            }

            j = next host number (j++; if j>N then j = 0);
        }
    }
}

If the sending of the LET entry package is successful, the function
delete_let(grp_let, j, let_table) deletes all the LET entries that
belong to both grp_let and let_table. Otherwise no deletion takes place and
this LET entry package will be sent again in the next round. If the local host is i, and
the remote host is j, then we denote the LET entry package obtained in algorithm 5.2 as
LET$_i^j$.

The second function periodically rolls back all entries in the NRT table. If a roll-
back is successful, the entry is deleted. The algorithm follows:
Algorithm 5.3.

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

The third function receives LET entry packages from the other RM servers and processes them according to the local UST table. Suppose the local host is \( j \) and one of the remote hosts is \( k \). If a RPC call issued by host \( j \) is performed by the destination host \( k \), then the destination host \( k \) will finally send this message back through the LET entry package extracted from \( LET_k \) table (see Property 5.3.4 in Section 5.3.3 later). Now host \( j \) will check its own \( UST_j \) entries against the received \( LET_k \) entries. Consider that we received the LET entry package \( LET_k^j \) from host \( k \). If any entry is found in both \( LET_k^j \) and \( UST_j \), that means the call returned \( US \) but was actually executed by host \( k \). So it is put into the \( NRT_j \) table for rolling back because the client has aborted the processing after it received the \( US \) return. If a call has returned \( US \) but was not executed by the destination host, there will be no such entry in the \( LET_k^j \) package received. In that case, we need to find the largest issuing time \( T \) of all RPCs executed by host \( k \) and issued by host \( j \), by joining the \( UST_j \) table and \( LET_k^j \) table together. Then we can delete those entries \( b : UST_j \ for \ b.host_to = k \ and \ for which \ the issuing time plus \ the maximum delay time \) (see assumption (2) of Section 5.3.3) is larger than the largest issuing time \( T \) found above. The corresponding algorithm follows:
Algorithm 5.4.

listenExternLets(ust_table, nrt_table)
UST *ust_table[];
NRT *nrt_table[];
{
    LET *rcvd_let[MAXENTRIES], *b;
    UST *c;
    int k;
    long T;

    while (TRUE) {
        /* listen to the sending of LET entry packages. */
        /* k is the host number which sends the package */
        /* k = receive(rcvd_let); */

        /* processing the executed RPCs on host k */
        /* put them into NRT if they appear in local UST */
        for all b ∈ rcvd_let {
            if (b.rpc = c.rpc where c ∈ UST) {
                NRT += b.rpc; /* += here means insert. */
                delete c from UST;
            }
        }

        /* processing other RPCs to host k and return US */
        /* delete them from UST if they are too "old"
        'not executed) */
        /* T = largest time of executed RPC */
        T = max(c.t | c ∈ UST, b ∈ rcvd_let and b.rpc = c.rpc);
        for all c ∈ UST {
            if (c.t ≤ T + MAX_DELAY) and (c.host_to == k)
                /* delete entries whose RPCs are not executed */
                delete c from UST;
        }
    }
}

The fourth function of the RM server manages the single RPC transactions of its clients. If any client program wants to make a RPC call, it calls the manage_rpc_calls() function and hands the call to the function. The function makes the RPC call. If the call is successful or not executed, it simply returns this fact to the client program. If the call returns the US value, then a UST entry is created and stored into the local UST. The algorithm follows:
Algorithm 5.5.

manage_rpc_calls(ust_table)
UST *ust_table[];
{
    int k;
    long t;
    UST *b;

    while (TRUE) {
        /* listen to local RPC call p */
        receive(p);

        /* do the RPC call */
        s = c(p);

        k = the destination host number;
        t = the time when issuing p;

        switch {
            case (s == OK): /* The RPC executed and returns OK */
            break;
            case (s == FL): /* the RPC is not executed */
            break;
            case (s == US): /* the RPC may or may not executed */
                initialise b;
                b.rpc = p; b.host_to = k; b.time = t;
                UST += b;
        }
        tell the client the RPC returns s;
    }
}

The fifth function listens to the local servers and logs the name of the RPCs executed locally into the local LET. That is, each server reports if it has performed a RPC. The two-phase protocol [Lampson81] can be used here to ensure that when a RPC has been executed by a server, the report from the server will arrive the RM server even in the presence of failures. The algorithm follows:
Algorithm 5.6.

```c
listen_local_servers(let_table)
LET *let_table[];
{
    char *p;
    int h;
    LET *b;

    while (TRUE) {
        /* listen to the report of executed RPC p */
        /* h is the reported client host number */
        h = listen_server(p);

        initialise b;
        b.rpc = p; b.host_from = h;
        LET += b;
    }
}
```

5.3.3. Properties

Before describing and proving several properties of the model, we make the following assumptions. It is easy to see that these assumptions are more or less realistic.

(1). The life-cycle of the system entities (hosts, links, and servers) is

```
loop
    work;
    crash;
    repair and restart;
end loop
```

Without loss of generality, we assume that the work time and down time (including crash, repair and restart time) are finite, and are denote as $T_w$ and $T_d$, respectively.

(2). The maximum delay time of a RPC call is MAX_DELAY. That is, if a RPC call is issued at time $t$, then either an OK or a FL value will return before the time $t + MAX_DELAY$, or a US value will return after this time.
(3). Sending (receiving) a LET entry package and making a rollback operation are all implemented by remote procedure calls. Hence we assume that their average execution time is the same as an ordinary RPC call.

(4). The probability that a RPC or a rollback call returns OK is much larger than the probability that it returns FL and US. If we denote by \( p \) the probability of a RPC or a rollback call returns OK, and denote by \( p \) the probability of other two returns, then we have \( p + q = 1 \) and \( p > 0.5 \). That is, a RPC or a rollback call will be successful in most cases if we view them over a sufficient long time period.

The following properties can be established for our single RPC transaction management model. We provide informal proofs.

**Property 5.3.1.** If a RPC returns OK, it has been executed and it will not be rolled back.

*Proof:* By definition, the RPC is executed correctly when OK returns. Rollbacks are only performed in the `rollback_nrt()` algorithm according to entries in the NRT. In turn, entries in the NRT are copied from the UST by the function `listen_extern_lets()`. The function `manage_rpc_calls()` ensures that, if a RPC returns OK, it will not be put into the UST, and so will not be put into the NRT. So the property is true.

**Property 5.3.2.** If a RPC returns FL, it has not been executed and it will not be rolled back.

*Proof:* Similar to the proof of Property 5.3.1.

**Corollary 1.** Any entry of LET resides in the LET for a finite time.

*Proof:* The function `send_my_let()` is responsible of sending out and delete LET
Chapter 5. RPC Transaction Management

table entries. Because we grouped together all LET entries which have the same
`host_from` field before sending them out, so we can view the LET table as having
the structure indicated in Figure 5.6:

Here *N* denotes the number of hosts in the system. Now, Algorithm 5.2 uses a loop
to send out entries, with one LET entry package each iteration. If the time needed
to send out a LET entry package is 1 time unit, then the time needed to complete
the loop has an upper bound of *N* units (we ignore the local processing time
because it is very small compared with the communication time). That is, if no
failures occur, the upper bound of sending out of any entry of the LET table is *N*
time units. As we have assumed that the maximum down time for a host is *T_d*,
that means the upper bound for sending out the LET entry package of a failed host
is *T_d* + *N*. So, any LET entry will be sent out in finite time. From now on, we
denote this upper bound as `SEND_LET_TIME`.

**Corollary 2.** The size of any LET entry package is finite.

**Proof:**

Algorithm 5.6 writes entries to the LET structure, while Algorithm 5.2 deletes
them. Let us assume that the time needed to send out a LET entry package is 1
time unit and that the average numbers of RPCs performed by a host in a time unit is \( s \). The worst case is that all the RPCs are called from one host \( i \). Corollary 1 tells us that the upper bound of sending out any entry of LET is \( N \) when no failures occur, so the maximum length of the LET entry package for host \( i \) is \( s \cdot N \). If host \( i \) fails, or the links to the host \( i \) fail, then no RPC calls from host \( i \) can be successful. That means the length of any LET entry packages for host \( i \) will not grow until the failures are recovered. Other cases are the same. So, the corollary holds.

**Corollary 3.** All RPCs that to be rolled back will be rolled back within finite time.

*Proof:*

According to our model, any RPCs call that to be rolled back are stored in the NRT. The system then is responsible of rolling them back by using Algorithm 5.3. We prove this corollary by two steps. At first we prove that the NRT will not grow indefinitely, then we prove that any entry in the NRT will be deleted (rolled back) within finite time.

Rollback and the deletion of NRT entries is the responsibility of function `rollback_nrt()`. At the same time, the RM server writes entries to NRT from UST using function `listenExternLets()`. In turn, the UST is written to by function `manageRpcCalls()` when RPCs return `US`. From assumption 4, only a small portion of RPCs need roll back, So, if on average there are \( s \) RPCs performed in a time unit, then the NRT is filled at a rate of \( s \cdot q \), where \( q \) is the probability that a RPC fails and, according to assumption 4, \( q < 0.5 \). In turn, on average there can be \( s \) RPCs rolled back per time unit because of assumption (3). That means there can be \( s \cdot p \) rollbacks return `OK` in a time unit, where \( p > 0.5 \). Because \( s \cdot q \) is the speed of filling the NRT and \( s \cdot p \) is the speed of deleting NRT and \( s \cdot q < s \cdot p \), the NRT will never grow indefinitely (please note that the \( p \) and \( q \) include both cases that the system is normal and abnormal, so we do
not need consider these cases separately).

If no failures occur, on each time the rollback operation \( r(b) \) of function rollback_nrt() will success and \( b \) will be deleted from the NRT. As we have proved that the NRT has a finite size, the for loop of function rollback_nrt() will finally go through all entries of the NRT. We may denote the upper bound time of this traverse as \( Q \). Now suppose host \( i \) is down, then any rollback operations to host \( i \) will not return \( OK \). As we have assumed that the maximum down time is \( T_d \), it is evident that the upper bound time for these rollback operations being performed is \( Q + T_d \). We use ROLLBACK_NRT_TIME to denote this upper bond.

**Property 5.3.3.** If a RPC returns \( US \), it will eventually be rolled back if it has been executed.

**Proof:**

Consider a remote procedure call \( c(p) \), made from a client on host \( i \) to a server on host \( k \), and suppose it returns \( US \) value. According to the function manage_rpc_calls(), an entry will be put into UST table. We denote it as \( d \), and \( d.rpc = p, d.host_to = k, \) and \( d.time = t \). After a finite time period (at most SEND_LET_TIME by Corollary 1), host \( k \) will send all the names of performed RPCs (that were issued by host \( i \)) to host \( i \). Suppose now all RPCs from host \( i \) to host \( k \) are processed before this return. If \( p \) has been executed, then there is a \( b : LET^i_k \), such that \( b.rpc = p \), where \( LET^i_k \) is the set of all entries in \( LET_k \) with host name fields equal to \( i \) (that is, the LET entry package from host \( k \) to host \( i \)). In that case, according to function listenExtern_lets(), \( p \) will be put into NRT table. After a interval (at most ROLLBACK_NRT_TIME according to Corollary 3), the \( RM_i \) will issue the rollback operation \( r(p) \) in function rollback_nrt(). If \( p \) has not been executed, there will be no such \( b \) in \( LET^i_k \).
So, the second segment of function `listenExternLet()` will delete $d$ from UST table, and no rollback operation is needed.

Now suppose $p$ is performed just after $k$ sent out package $LET^i_k$. In that case, $b$ will be put into $LET^i_k$ but not in $LET^i_k$. Because the RPC returns US, $d$ is put into $UST_i$.

In function `listenExternLets()`, $RM_i$ will find that $d.time > T + \text{MAX\_DELAY}$ because of assumption 2. So, $d$ will remain in the $UST_i$ table.

Eventually, MAX\_DELAY time will pass and when host $k$ sends the $LET^i_k$ next time, $b$ will be there and will be rolled back as above.

Because the round time periods for function `sendMyLet()` and function `rollbackNrt()` are finite, the MAX\_DELAY time is finite, and the time between two neighbour works for system entities are finite, the above rollback operation will eventually take place.

**Property 5.3.4.** The LET, NRT, and UST structures will not grow indefinitely and any entry will be sent out or deleted eventually.

**Proof:**

From Corollary 1 and 2, the finite number of hosts in the system and the limited down time, it is easy to see that the assertion for the LET is true.

The UST is written to by function `manageRpcCalls()`. It is easy to see that the average length of the UST is less than or equal to the average length of the LET, because the UST contains only those RPC entries with return values US, while the LET contains the OK return entries. By assumption 4 the assertion holds.

From the assumption that the down time of any system entity is finite, we know that all RPCs issued by a host will be acknowledged and checked in function `listenExternLets()`. So we can conclude that any entry in the table will be deleted eventually.
From Corollary 3 and its proof, it is easy to see that the assertion for NRT table is true.

5.4. Parallel RPC Transaction Management

In the last section, we have established the system for single RPC transaction management. In this section, we extend the above model and build the parallel RPC transaction management system.

5.4.1. Algorithms

The RPC manager and its algorithms used to process parallel RPC transactions are almost the same as those used to process single RPC transaction calls. The single difference is that the function `manage_rpc_calls()` is extended to include the parallel RPC transaction processing case.

The extended function receives the components of a parallel RPC transaction call from the local host and processes the calls within the parallel RPC transaction concurrently. If all the RPCs return `OK`, or if all the RPCs return `FL`, then it simply tells the client the result and exits (to receive a new transaction). If there are any `US` returns, or if there are `FL` and `OK` returns, by the semantics of the parallel RPC transaction, the transaction must be aborted and any of its executed RPCs must be rolled back. So, the function will write some relevant entries into the NRT table and let the system to roll these executed RPCs (or suspected executed RPCs) back. Next is the extended function:
Algorithm 5.7.

manage_rpc_calls(ust_table)
UST *ust_table[];
{
UST *b;

while (TRUE) {
    /* listen to local parallel RPC transaction calls */
    receive(T = { p1, ..., pm });

    /* concurrently execute the RPC calls within the transaction */
    COBEGIN
        ret_i = c(p_i),
        k_i = the destination host number,
        t_i = the time when issuing c(p_i),
        i = 1, 2, ..., m;
    COEND;

    SU = {p_i | ret_i == US};
    SF = {p_i | ret_i == FL};
    SO = {p_i | ret_i == OK};

    switch {
        case (SU == SF == ∅):
            /* all RPCs executed and returned OK */
            tell the client the transaction returns OK;
            break;
        case (SO == SU == ∅):
            /* error, but none RPCs are executed */
            tell the client the transaction returns FL;
            break;
        case (SU ≠ ∅ || (SF ≠ ∅ && SO ≠ ∅)):
            /* error, but some RPCs maybe executed */
            if (SU ≠ ∅) {
                for each p_i ∈ SU {
                    b.rpc = p_i; b.host_to = k_i; b.time = t_i;
                    UST += b;
                }
            }

            if (SO ≠ ∅) {
                for each p_i ∈ SO
                    NRT += p_i;
            }
            tell the client the transaction returns US;
            break;
    } /*switch */
} /* while */
Chapter 5. RPC Transaction Management

5.4.2. Properties

It is evident that Property 5.3.4 holds for parallel RPC transactions. The following properties can be easily established:

Property 5.4.1. If a parallel RPC transaction returns OK, all its RPCs have been executed and will not be rolled back.

Proof:
The only place for the system to return OK is when $S_{US} = S_{FL} = \emptyset$. In that case, all RPC are executed correctly, and nothing is put into UST table. So, they will not be rolled back.

Property 5.4.2. If a parallel RPC transaction returns FL, no RPC of the transaction is executed.

Proof:
The transaction returns FL only if $S_{OK} = S_{US} = \emptyset$. That is, all the RPCs of the transaction return FL. So, no RPC of the transaction has been executed.

Property 5.4.3. If a RPC transaction returns US, any executed RPCs will be rolled back.

Proof:
The transaction returns US only if $S_{US} \neq \emptyset$ or $(S_{FL} \neq \emptyset$ and $S_{OK} \neq \emptyset)$. If $S_{US} \neq \emptyset$, Algorithm 5.7 then builds UST entries for all RPCs that return US and inserts them in the UST. According to Property 5.3.4, those entries will be put into the NRT if their RPC have been executed. Again by Property 5.3.4, entries of the NRT will be rolled back eventually.
Chapter 5. RPC Transaction Management

If \( S_{FL} \neq \emptyset \) and \( \forall_{OK} \neq \emptyset \), Algorithm 5.7 then builds NRT entries for all RPCs that return OK and insert them into the NRT. Again by Property 5.3.4, they will be rolled back eventually.

5.5. Remarks

The design of a RPC transaction manager is described in this chapter and a number of its basic properties are indicated. After an introduction to the problem and to the RPC transaction models, the RPC transaction management problem was addressed in two stages. At first, a system for managing transactions involving a single RPC call was developed. The relevant algorithms and properties of the system were described. Then, by extension, a system for managing parallel RPC transactions was developed.

In the model under study, all RPCs are capable of being rolled back, and our system ensures that a RPC or a set of parallel RPCs will be executed if no errors occur. In addition, any executed components of the transaction will be rolled back if something goes wrong.

Almost all the existing transaction management approaches use a two-phase protocol. But if applying a two-phase model to the RPC-oriented transaction management, the efficiency of such model will be less than our proposed system, mainly because the execution checking (checking if a RPC will return OK or not) will take a great deal of time, and the stable storage management in a two-phase protocol is more complex than our model.

Our model can act as a run-time system within the programming environment. It is transparent to programmers. Programmers will not have too much burden to maintain the RPC transactions in their programs. They can use RPC transaction calls as usual RPC calls and the system will do all the job. We feel this is better than the language level implementation.
Several extensions of the model is possible. For example, one may want to explore the nested RPC transaction model, or extend the model to maintain the atomic RPC transactions.
Chapter 6
Conclusions

6.1. Summary

In this thesis we have designed and implemented several network services for distributed computing. It is suggested that these services contribute to the following goals.

1. *Easing the burden of distributed programming.* In Chapter 2, we report the design and implementation of a set of rapid prototyping tools for distributed programming. Currently three tools are provided. They are: a distributed frame generator, a user interface generator, and an application module generator. A user provides these generators with appropriate specification files, and an executable version of the distributed program is directly generated. These tools can greatly reduce the burden of distributed programming. Firstly, they reduce the time from the design to an executable version of a distributed program. Secondly, they reduce the error possibilities in programming because the generators have been tested and debugged. Thirdly, they can easily accommodate design changes during the development process.
2. **Provision of tools for distributed debugging.** In Chapter 3 we describe the design and implementation of a distributed monitor which is of help in the debugging of distributed programs. The monitor is event-based. It records all interesting events into the monitor’s database. A partial ordering can be established between these events and the user can use this partial ordering to trace and replay the program’s execution. This can help a user to locate the bugs of a distributed program. The monitor and its database is distributed. It can be used to monitor several programs simultaneously.

3. **Provision of tools for execution time estimation.** Chapter 4 describes an execution time evaluation model for distributed programs. A nondeterministic algorithm is provided for estimating execution time of complex distributed programs. Then the explicit solution for some special cases is developed. After some simplification, the lower and upper bounds of execution time of complex distributed programs are defined. Appropriate calculation and simulation tools are implemented according the model.

4. **Provision of services for managing RPC transactions.** Chapter 5 proposes a system to manage such transactions. At first, a RPC model appropriate to our purpose is described. Then we design a transaction manager for single RPC calls, which can roll back a RPC call if something goes wrong during the call. By extending the single RPC transaction manager, a parallel RPC transaction manager is designed for managing a group of parallel RPC calls. Some properties of these managers are also described.

### 6.2. A Computer Science Perspective

This dissertation may contribute to one of the steps by which distributed computing becomes commonplace. A decade ago, researchers concentrated on the interconnect-
Chapter 6. Conclusions

The enabling technology for distributed software is almost mature. So, it is time to make the distributed programming less painful, to make distributed programs easy to debug, to provide tools for better distributed program performance evaluation, to make distributed programs more reliable and so on.

We believe that one way to make distributed computing commonplace is to provide a range of network services that overcome the difficulties encountered and that meet the new needs that emerge. The design and implementation described in this thesis contributes to this effort.

6.3. Future Research Directions

In the short term, the services provided here can be refined and extended. For example, the distributed frame generator needs to accommodate other kinds of RPC facilities instead of only NCS/RPCs. The user interface generator needs to use more sophisticated window systems. The application module generator needs to be extended to generate many other possible application modules instead of only database-oriented applications. More effort is needed to let the prototyping tools help distributed algorithm design and implementation.

In distributed debugging, we need to combine our distributed monitor with a sequential debugger to facilitate the location of program bugs. We also need a more sophisticated user interface for the monitor to replay events and associated messages. Some efforts are also needed to debug time-strict distributed programs.

The execution time estimation tool may be refined by exploring better upper and lower bounds, or even an explicit solution for more general distributed programs than those presented. Another possible refinement is the extension of the distributed program
Chapter 6. Conclusions

definition to include more complex control structures, such as selection and loop.

In RPC transaction management, we need to implement the design and to test the ideas presented in this dissertation. Another valuable option would be to explore the possibility of a nested RPC transaction model as well as of maintaining the full RPC transaction atomicity.

In the medium and long terms, more sophisticated network services should be implemented to overcome other difficulties, such as load balancing, mapping distributed problems onto a known architecture and so on, and to meet emerging needs. Where it is possible, an integrated distributed computing environment is also desirable.
Appendix A

The following document is a report written while the author was visiting the Research and Development Laboratory of Apollo Computer Inc. in Chelmsford, MA, USA. It describes a distributed program system that produces a calendar/diary service for a pool of clients.
1. Problem

We define:

**Caller:**
A person that issues meeting data (see below).

**Callee:**
one of the persons that a caller wants to have meeting with.

**Meeting-data:**
A message which includes caller’s name, callee’s name(s), meeting place, meeting time interval, and other related information.

The program is used to solve the following problems:

1. Given a meeting-data, we want it to be arranged in each person’s (caller and callee(s)) calendar database if the meeting place is available and all persons in that meeting are free during the meeting interval.

2. Allow a caller to delete a meeting which he/she entered from all participants’
Appendix A

calendar database.

(3). Allow a person to set some reminders which are just for himself to use.

(4). Automatically delete all old meeting entries (that is, the meeting date is several days ago).

2. Structure of DISCAL

There are three kinds of database files in DISCAL:

(a). A Name Database. It contains the names and their calendar database UUIDs of all legal users of the program. The UUIDs are used to locate the calendar databases (see below) of every user through GLB.

(b). A Meeting-Room Database. If a meeting room has been booked for a meeting for a time period, there will be an entry in this database which contains the room name and meeting time interval.

(c). Calendar Databases. If a user has a meeting, there will be an entry in his calendar database which contains the user name, meeting time interval, meeting place, caller of the meeting, and all names of the participants of the meeting. Virtually there is one calendar database for each legal user. For the purpose of easy management, we put the calendars of a group of people into a single database file. For example, the group can be all persons within a department, within a research group, or even within a local network. There is almost no limit about the size of one database file. The only concern is that, the larger the database file is, the slower the program speed will be. At this time, ten (numbered 0 to 9) calendar database files can exist within the system. We call these numbers as calendar server numbers (see below).

For each database file, there is one server which maintains it. We call them Register Server, Room Server, and Calendar Server(s), respectively. If a user's name cannot be found in the name database, the program now will use calendar server 0 (that is, the
calendar database file managed by calendar server 0) as default calendar database and automatically adds the name entry into the name database.

![Diagram](image)

**Figure. A-1. The architecture of the distributed calendar system**

When you invoke the program, your login name is used by the program as a key to look for your calendar database UUID from name database. This UUID is then used to check your calendar database to see how many entries you have there and the result is reported to you. If you issue a meeting (by providing a meeting-data), the program first checks the meeting-room database to see if the meeting room is free for the meeting time interval. If it is free, then the calendar database UUIDs of all callees are found from the name database and their calendar databases are checked to see if all of them are free during the meeting time interval. If the above check passes, a proper entry will
be inserted into the meeting-room database and each related calendar database.

One problem will arise: How can we control the concurrent accesses to database files? For example, if two users want to have a meeting with the third person contemporaneously, who will win the competition and how can we avoid deadlocks during the competition? We will talk about this in next sections.

The structure of the Distributed Calendar is as in Figure A-1 (there may be more calendar database files involved. But only two are pictured for simplicity).

3. System Consideration

NCS is the basis of this program. All database files are managed by servers which export some remote procedures that can be called by client programs.

The DAM (Data Access Manager) ([Perry89] and [Martin88]) is used to maintain all database files. By using this product, one can insure that at any time there is only one user can update (delete, add) the database. Other update operations are queued by the program. There can be more than one read-only operations at any time.

There is no competition problems related to name database. So from now on, we will omit it from our discussion unless we mention it explicitly.

There are following possible status for each entry in a meeting-room or calendar database:

BOOKED:

The time interval [startval, endval] has been booked for meeting and cannot be allocated again. Where "startval" is the start date and time of the meeting and "endval" is the end date and time of the meeting;

TEMP_BOOKED:

The time interval [startval, endval] is temporarily booked by a caller. It may be re-allocated to another meeting after it is set free;
AVAILABLE:

We say a time interval \([s, e]\) of a user/room is available if for any \(t : [s, e]\), and any \(q : [s_i, e_i], i : \) all \{BOOKED and TEMP_BOOKED entries of that user/room\}, we have \(t \neq q\). Because this can be decided by algorithm, so we delete the entry when its time interval becomes available instead of set its status to AVAILABLE.

4. Structure of client programs

The following algorithm is used in the client program for booking a meeting:

(a). Display the program menu. If the user chooses the "Enter meeting" selection, ask the user (now he is the caller of the meeting) input meeting data.

(b). Ask the room server to temporarily book the meeting time interval of the meeting room. If successful, an entry of this room with status TEMP_BOOKED is added into the room database. If not successful (that is, the time interval of that room is not AVAILABLE -- being BOOKED or TEMP_BOOKED), return to step (a).

(c). Ask the calendar server of the caller’s calendar database to temporarily book the meeting. If successful, an entry of the caller with status TEMP_BOOKED is added into his calendar database. If not successful (that is, the time interval of the caller is not AVAILABLE), cancel the previous meeting book (that is, delete the entry added by step (b) from the room database) and return to step (a).

(d). Ask the calendar servers of all callees’ calendar databases to temporarily book a meeting. If successful, an entry of each callee with status TEMP_BOOKED is added into his calendar database. If not successful (that is, the time interval of any callee is not AVAILABLE), cancel the previous meeting booking. Cancel the caller’s meeting booking. And cancel any callee’s meeting books which have been added during this step. Return to step (a).

(e). Confirm all TEMP_BOOKs we just made (room, caller, and all callees) by changing their status from TEMP_BOOK into BOOKED. Meeting is booked success-
Appendix A

fully. Return to step (a).

Please notice that the above algorithm may not obtain the best meeting time interval. For example, if there are two callers A and B. Caller A wants to have a meeting with callee C and D at time 10:00-11:00, while caller B wants to have a meeting with callee C at time 10:30-11:30. Suppose both are at the same date. Also suppose A, B and C are AVAILABLE for the meeting time interval, but D is not AVAILABLE for 10:00-11:00. Now, if caller A goes first and TEMP_BOOKED callee C. Then caller B found C is TEMP_BOOKED and has to return step (a) for another try by asking another meeting data. But A will discard C’s temporal book when he finds that D is not AVAILABLE. So, when B returns to step (a), callee C may be AVAILABLE for B’s meeting again and B may miss it!

We can improve the above algorithm by adding some waiting loops. First we can define a new status:

PARTIAL_AVAILABLE:

We say a time interval \([s, e]\) of a user/room is \emph{partial available} if there is a \(t : [s, e]\), and a \(q : [si, ei]\), \(i : \) any one of \{TEMP_BOOKED entries of that user/room\}, that \(t = q\).

It is easy to know from the algorithm that if a time interval \([s, e]\) is PARTIAL_AVAILABLE, then for any \(t : [s, e]\), and any \(q : [si, ei]\), \(i : \) all \{BOOKED entries of that user/room\}, we always have \(t \neq q\). This is because that the time intervals of all entries in the database for a user/room are disjunctional.

The method for improving the algorithm is as follows. At each time, when we find the time interval is not AVAILABLE because it is PARTIAL_AVAILABLE, we can wait for a while and try again. Only after, say, three times re-try that we may abandon the meeting and return to step (a).

It is easy to prove that the above algorithm is deadlock free. The original algorithm is deadlock free because the time intervals occupied by a caller (that is,
TEMP_BOOKED by that caller) are abandoned as long as he finds that any of his requirement is not satisfied. The improved algorithm is also deadlock free because it only waits and tries a limited time for a temporarily booked time interval. After that he will abandon all the occupied time intervals.

5. About server program

As mentioned earlier, the program uses the DAM to maintain all database files. For each server, there are the following remote procedures that can be used by any of its client:

(a). Find an entry by using the "primary key";

(b). Find all entries by giving the key number and key (This includes function (a). But function (a) is more efficient if finding by primary keys).

(c). Add a new entry into the database.

(d). Change the contents of an existing entry.

(e). Delete an existing entry.

For room and calendar servers, there also is a procedure which is used to delete all old entries. Now, an old entry is defined as an entry which start time is seven days before today. It is easy to change the definition. There are some other remote procedures which are supported by the servers and are used by the management tools. We will not discuss them here.

A server program has two parts (name server only has the first part). The first part is the "real" server part. That is, it exports remote procedures by listening for client calls. The second part is another process which clears the database by deleting all the old entries. This part sleeps for a time interval which is provided by the user at the start of the program (or by default, one day) and then wakes up to do the cleaning.

We also provide for each kind of database server a management tool which can do general management for the database, such as enter, edit, delete and find an entry; show
all entries which have the same key; and even shutdown the server. We only suppose
the manager of the database will use the tool when necessary. No general users are sup­
posed to use it.

6. Usages

CA_PW_SER(1)

NAME

capw_ser - DISCAL calendar database server

SYNOPSIS

capw_ser <server No> <database file> [-v] [-clean]
[clean_time_interval]

DESCRIPTION

Ca_pw_ser is the calendar database management server. It maintains the calendar
database file which name is provided by the command line. One database file can
be managed by only one calendar server, otherwise, the second server will fail to
function. Also, one calendar server number (from 0 to 9) can be assigned to only
one calendar server, otherwise the previous invoked server will be substituted by
the last one (this is because in GLB, the old entry will be substituted by the new
one if a register call comes with the same UUIDs as the old one).

When ca_pw_ser is invoked, it splits into two processes. The parent process sleeps
for a period of time and then notifies the child server to do the cleanup. Then it
sleeps again. The child process is the real server process. It first registers itself to
the GLB and does all the jobs for RPC preparation. Then it opens the calendar
database file, prepares a signal function for cleanup, and listens to the client calls.
If the child process received the signal from the parent process, the cleanup func­
tion is invoked and all old entries in the database cleaned. If the child process
received the shutdown command twice, it closes the database file and signals the
parent process to kill both processes.

OPTIONS

-\*v\* Be verbose. In that case the server will output some messages which can help a user to know if the server is working well.

-clean If this option is used, the old entries will be cleaned each time interval. Otherwise no cleanups.

clean_time_interval

Time interval in minutes for the parent process of the server program to invoke the cleanup procedure.

COMMAND LINE INPUTS

<server No>

Calendar server number. As we have mentioned, ten calendar database files can exist. And for each file we need a different calendar server. These servers must have different UUIDs for registering to GLB. We use this number to get the proper UUID. So, be sure there is no calendar servers running with the same server number.

<database_file>

The file name of the database that the server is going to manage.

FILES

The related files are:

ca_pw.idl ca_global.h ca_pw_rpcfuncs.h ca_pw_ser.c dc_global.h
dc_util.c uuid_file.h

Also, all server programs include DAM including files (dam.h, dsm.h, and symtab.h) and are linked with "dds_tools_lib".

RM_PW_SER(1)
NAME

rm_pw_ser - DISCAL meeting room database server

SYNOPSIS

rm_pw_ser <database_file> [-v] [-clean] [clean_time_interval]

DESCRIPTION

Rm_pw_ser is the meeting room database management server. It maintains the meeting room database file which name is provided by the command line. Unlike the calendar database, there is only one meeting room database in the system. So one server is enough.

When rm_pw_ser is invoked, it splits into two processes. The parent process sleeps for a period of time and then notifies the child server to do the cleanup. Then it sleeps again. The child process is the real server process. It first registers itself to the GLB and does all the jobs for RPC preparation. Then it opens the meeting room database file, prepares a signal function for cleanup, and listens to the client calls. If the child process received the signal from the parent process, the cleanup function is invoked and all old entries in the database cleaned. If the child process received the shutdown command twice, it closes the database file and signals the parent process to kill both processes.

OPTIONS

-Be verbose. In that case the server will output some messages which can help a user to know if the server is working well.

-clean If this option is used, the old entries will be cleaned each time interval. Otherwise no cleanups.

clean_time_interval

Time interval in seconds for the parent process of the server program to invoke
Appendix A

the cleanup procedure.

COMMAND LINE INPUTS

<database_file>

The file name of the database that the server is going to manage.

FILES

The related files are:

rm_pw.idl  rm_global.h  rm_pw_rpcfuncs.h  rm_pw_ser.c  dc_global.h  dc_util.c

NI_PW_SER(1)

NAME

ni_pw_ser - DISCAL name database server

SYNOPSIS

ni_pw_ser <database_file> [-v]

DESCRIPTION

Ni_pw_ser is the name database management server. It maintains the name database file which name is provided by the command line. There is only one name database in the system. So one server is enough. Also because there is no cleanup involved, the server does not need to split into two processes now.

The server first registers itself to the GLB and does all the jobs for RPC preparation. Then it opens the name database file and listens to the client calls. If the server receives the shutdown command twice, it closes the database file and exits.

OPTIONS

-v Be verbose. In that case the server will output some messages which can help a user to know if the server is working well.

COMMAND LINE INPUTS
Appendix A

<database file>

The file name of the database that the server is going to manage.

FILES

The related files are:

ni_pw.idl  ni_global.h  ni_pw_rpcfuncs.h  ni_pw_ser.c  dc_global.h  dc_util.c

DISCAL(1)

NAME

discal - DISCAL program (client)

SYNOPSIS

discal [-v] [-a] [-name]

DESCRIPTION

Discal is the client part of DISCAL, and also the only program that an ordinary user can use. It uses UNIX system call to get the user’s login name and checks the proper calendar database to report to the user how many messages he has got. Please notice that, each time when the program wants to do any operation on a person’s calendar database, it has to know the location (the UUID of the database) and that may involve looking up the name database first. Only the user’s (that is, the caller of a meeting) calendar database location is kept and need not to lookup again. If the user’s name can not be found in the name database, and the option [-a] is not specified, this name will be added into the name database and the default calendar database (number 0) is used.

After all the preparation, the program displays a menu as follows ([ and ] are used to separate the program display with the text):
The Distributed Calendar supports following operations:

1. Enter Meeting: Enter a new meeting.
2. Enter Reminder: Enter a new reminder.
5. Quit:

Select one:

By selecting the numbers, a user can perform the job he wants the program to do. The first selection is to enter a meeting. If you choose this, the program asks you to input the participants (callees) of the meeting. Please notice that you do not need to input your name because it has been included automatically (that is, the user is the caller and also one of the participants). The maximum number of participants is 20 and the maximum name length is 19. Of cause one can easily change these. A blank line will terminate the callee list. Then the program displays the callee list and asks the user if he wants to make any change. The user can delete from or add to the list, and can change any entry in the list. If the list is ready, the program then asks the meeting’s date by providing a week’s date (also a selection for input any date) from today for selection. For example, if today is 7/6/89, the display will be:

[    1) Today    (7/6/89, Thursday)    
  2) Tomorrow  (7/7/89, Friday)    
  3) Saturday  (7/8/89)    
  4) Sunday    (7/9/89)    
  5) Monday    (7/10/89)    
  6) Tuesday   (7/11/89)    
  7) Wednesday (7/12/89)    
  8) None of the above

Choose one:
]

If you choose 8), the date format is

<month>/<day>/<year>

where <year> can be 4 or 2 digits. <month> should be 1-12. If the input is greater
than 12, its module number is used. <day> should be 1-31. Also the module number will be used if the it is greater than 31.

Next is the meeting time. Which includes the start and end times. Its format is:

<hour>[:<min>][a][p]

If the hour is less than 12 and there is no a/p specified, the program will ask you the time is morning or afternoon. If the end time is less than the start time, you will be asked re-input them again. Then the program asks you to input the activity of the meeting. This can be any comment you want to put into the meeting message (for example, the purpose of the meeting). The maximum length now is 149. The meeting room is the last information asked. After that, the program displays the meeting data and ask you to confirm the meeting data. If it is confirmed, discal does the meeting booking transparently. No matter the booking succeeds or fails, the program displays the result and returns to the menu.

The second selection is for entering a reminder. Because this is just for the user himself, there is no callee, date, and time input as above. The first asked information is the activity. You can put any comment here. Next is the place. If you do not need it, simply press RETURN. This message is only stored in the user’s calendar database.

The third selection is to show all the engagement (meetings and reminders) that the user has.

The fourth selection is used to delete an engagement. If the engagement is a reminder, then the only thing is to delete it from the user’s calendar database. If the engagement is a meeting, the program first checks if the user is the caller of the meeting. If not, the deleting is rejected. If the check passed, then it deletes the room booking from the meeting room database, and deletes the caller’s meeting entry from the user’s calendar database. After that, the program deletes every
callee’s meeting entry from that callee’s calendar database.

OPTIONS

-v Be verbose. In that case the server will output some messages which can help a user to know if the server is working well.

-a If you specify this option, your name will not be added into the name database if it is not in.

-name You can use this option to "cheat" the program. In this case, the program will ask you input a user name and use this name as the caller’s name.

FILES

The related files are:

discal.c dc_global.h dc_util.c

ni_pw.idl ni_dp.c rm_pw.idl rm_dp.c ca_pw.idl ca_dp.c
References


References


References


References


[Heidelberg83] P. Heidelberg and K. Trivedi, "Analytic Queueing Model for Pro-


References


References


References


References


References


References


References

Conference on Computing and Information, ICCI'90, pp. 117-121, Niagara Falls, Ontario, Canada, May 1990.


