This item is held in Loughborough University's Institutional Repository (https://dspace.lboro.ac.uk/) and was harvested from the British Library's EThOS service (http://www.ethos.bl.uk/). It is made available under the following Creative Commons Licence conditions.

C O M M O N S D E E D

Attribution-NonCommercial-NoDerivs 2.5

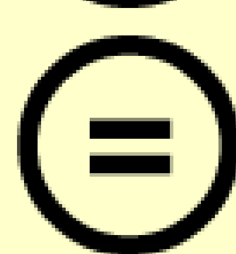
You are free:

- to copy, distribute, display, and perform the work

Under the following conditions:

Attribution. You must attribute the work in the manner specified by the author or licensor.

Noncommercial. You may not use this work for commercial purposes.



No Derivative Works. You may not alter, transform, or build upon this work.

- For any reuse or distribution, you must make clear to others the license terms of this work.
- Any of these conditions can be waived if you get permission from the copyright holder.

Your fair use and other rights are in no way affected by the above.

This is a human-readable summary of the Legal Code (the full license).

Disclaimer

For the full text of this licence, please go to: http://creativecommons.org/licenses/by-nc-nd/2.5/

Supporting Distributed Computation over Wide Area Gigabit Networks

by

Jon P. Knight

B.Sc. (Hons.), Loughborough University of Technology (1991)

A Doctoral Thesis

Submitted in partial fulfilment of the requirements

for the award of

Doctor of Philosophy of the Loughborough University of Technology

September 1995

© by Jon P. Knight, 1995

Supporting Distributed Computation over Wide Area Gigabit Networks

by

Jon P. Knight

Submitted to the Department of Computer Studies
on September 8th 1995, in partial fulfillment of the
requirements for the degree of
Doctor of Philosophy

Abstract

The advent of high bandwidth fibre optic links that may be used over very large distances has lead to much research and development in the field of wide area gigabit networking. One problem that needs to be addressed is how loosely coupled distributed systems may be built over these links, allowing many computers worldwide to take part in complex calculations in order to solve "Grand Challenge" problems. The research conducted as part of this PhD has looked at the practicality of implementing a communication mechanism proposed by Craig Partridge called *Late-binding Remote Procedure Calls* (LbRPC).

LbRPC is intended to export both code and data over the network to remote machines for evaluation, as opposed to traditional RPC mechanisms that only send parameters to pre-existing remote procedures. The ability to send code as well as data means that LbRPC requests can overcome one of the biggest problems in Wide Area Distributed Computer Systems (WADCS): the fixed latency due to the speed of light. As machines get faster, the fixed multi-millisecond round trip delay equates to ever increasing numbers of CPU cycles. For a WADCS to be efficient, programs should minimise the number of network transits they incur. By allowing the application programmer to export arbitrary code to the remote machine, this may be achieved.

This research has looked at the feasibility of supporting secure exportation of arbitrary code and data in heterogeneous, loosely coupled, distributed computing environments. It has investigated techniques for making placement decisions for the code in cases where there are a large number of widely dispersed remote servers that could be used. The latter has resulted in the development of a novel prototype LbRPC using multicast IP for implicit placement and a sequenced, multi-packet saturation multicast transport protocol. These prototypes show that it is possible to export code and data to multiple remote hosts, thereby removing the need to perform complex and error prone explicit process placement decisions.

Thesis Supervisor: Dr. Stephen P. Guest
Title: Lecturer

Thesis Co-supervisor: Dr. Ian A. Newman
Title: Reader

Contents

1 Introduction **9**

1.1 Introduction 9

1.2 The Latency Problem 11

1.3 Points Addressed 11

1.4 Outline of Thesis 13

2 Survey of Networking and Distributed Systems **14**

2.1 Introduction 14

2.2 The Gigabit Revolution 15

2.2.1 Protocol Processing 16

2.3 High Speed Networking Testbeds 19

2.3.1 US Testbeds 19

2.3.2 European Testbeds 21

2.4 The Need for WADCS 23

2.5 Communication Mechanisms 24

2.5.1 Message Passing 24

2.5.2 Traditional Remote Procedure Calls 25

2.5.3 Distributed Shared Memory 26

2.5.4 Remote Execution of Programs 27

2.5.5 Remote Job Entry 27

2.5.6 Process Migration 28

2.5.7 Distributed Filesystem Access 28

2.5.8 File Transfer Protocols . . . 29
2.6 An Overview Late-Binding RPC . . . 30
2.7 Potential Problems with LbRPC . . . 32
2.8 Summary . . . 34

3 Intermediate Code **35**
3.1 Introduction . . . 35
3.2 Requirements for Intermediate Exportable Code . . . 36
3.3 Partridge's Suggested Representations . . . 38
3.3.1 PostScript . . . 38
3.3.2 HEMS . . . 38
3.3.3 LISP . . . 39
3.4 Attempted Universal Languages . . . 40
3.4.1 Compiler oriented languages . . . 40
3.4.2 Abstract Machines . . . 43
3.4.3 Universal High Level Languages . . . 44
3.4.4 Interface Definition Languages . . . 47
3.5 The ANDF Approach . . . 47
3.5.1 Structure of ANDF . . . 48
3.5.2 Pros and Cons of ANDF usage with LbRPC . . . 49
3.6 Avoiding the UNCOL Problem . . . 51
3.6.1 Multiple Intermediate Languages . . . 52
3.6.2 Limiting Intermediate Code Functionality . . . 53
3.7 Dynamic Code Generation . . . 56
3.8 Compilation vs. Interpretation . . . 57
3.9 Summary . . . 59

4 Exporting Data Structures **61**
4.1 Introduction . . . 61
4.2 Existing Representations . . . 62
4.2.1 Early Work . . . 62

4.2.2 Courier and Cedar 62
4.2.3 XDR 63
4.2.4 ASN.1 64
4.2.5 Minimalism in Sprite 65
4.3 Problems with External Representations 66
4.4 Conclusions 68

5 Placement Decisions **70**
5.1 Introduction 70
5.2 Network Performance Measurement 72
5.3 Process Placement in Tightly Coupled Systems 73
5.4 Optimising Network Usage 75
5.5 WAN Performance Measures 76
5.6 Dynamic Behaviour 77
5.7 Strawman Proposal 79
5.7.1 SNMP and ICMP Probing 80
5.7.2 Problems 82
5.7.3 Using Heuristics 83
5.8 Implementation 84
5.9 Multicast Placement 86
5.10 Multicast LbRPC 88
5.11 TCL-DP Prototype Implementation 89
5.11.1 Basic Multicast Hooks 89
5.11.2 Multicast LbRPC Mechanism 90
5.11.3 Performance 92
5.12 Discussion 96
5.13 Conclusions 99

6 Transport Protocols **101**
6.1 Introduction 101
6.2 Previous Multicast Transport Protocols 102

6.2.1 Unreliable vs. Reliable 102
6.2.2 Positive Acknowledgments 103
6.2.3 Negative Acknowledgments 104
6.2.4 Saturation Protocols 105
6.2.5 Fault Tolerance 105
6.2.6 Lessons Learnt 106
6.3 Designing an MTP to support MLbRPC 106
6.3.1 Acknowledgments, Windows and Saturation 107
6.3.2 Return Path 109
6.4 MADTP Implementation 110
6.4.1 Sequence number space 110
6.4.2 Transaction Identifiers 111
6.4.3 Header Format 111
6.4.4 Receiving a MADTP Packet 113
6.5 Performance 117
6.6 Discussion and Conclusions 118

7 Security Considerations 120

7.1 Introduction 120
7.2 Security Classifications 121
7.3 Securing Electronic Mail 123
7.4 Security in Distributed Systems 125
7.5 Implementing Security in LbRPC 128
7.5.1 Simple Authentication 129
7.5.2 Asymmetrical Digital Signatures 130
7.5.3 Dual Encryption 135
7.5.4 The Multicast Environment 136
7.5.5 Performance 136
7.6 Payment 137
7.7 Conclusions 139

8 **Discussion and Conclusions** **141**
8.1 Introduction 141
8.2 Review of limitations 141
8.2.1 Intermediate Code Representations 141
8.2.2 Process Placement 143
8.2.3 Lower Layer Issues 144
8.2.4 Security 145
8.3 Discussion 146

9 **Appendices** **148**
9.1 Appendix A: Initial Simulation Results of Late Binding against Traditional RPC 148
9.1.1 Introduction 148
9.1.2 Effect of communications delay on total execution time 149
9.1.3 Bandwidth Effects on Total Execution Time 151
9.1.4 Cost of number of traditional RPC requests 153
9.1.5 Conclusions 153

10 **Glossary** **155**

Acknowledgements

This thesis would not have been possible without the help and support of a number of people. Firstly I must thank Dr Guest and Dr Newman, my supervisors, for giving me the chance of doing this research and for putting up with me as a research student for over three years. Martin Hamilton, Anne Bricis, Barbara Dobson, Richard Butterworth, Jon Allen and Dave Temple also deserve thanks for providing sounding boards for ideas and for letting me rant and rave at times.

Lastly I'd like to acknowledge the support that my parents have given me throughout my educational career. Without their help and encouragement as a child I would not have experienced the joy I now find in seeking knowledge and asking questions. This thesis is dedicated to both of them. It is but a small return for what they have done for me, but it is given with love.

Chapter 1

Introduction

1.1 Introduction

The network environment used to support distributed computations is undergoing a remarkable change. Distributed systems have traditionally been most widely deployed over high speed Local Area Networks (LANs), leaving the slower and more error prone Wide Area Network (WAN) links for electronic mail, file transfer and interactive login applications. The latencies experienced over WAN links along with the greatly reduced bandwidth mean that computations spread over a wide area are slowed down significantly by any network transits they are forced to make. Many distributed systems paradigms designed with the LAN environment in mind assume that network transits are relatively cheap in terms of the equivalent number of machine instructions that can be processed while a message is in transit over the network. This mismatch between the design assumptions of the distributed computation mechanisms and the realities of the WAN environment has limited the introduction of distributed systems that can operate efficient over widely geographically dispersed machines.

The revolutionary force that is set to alter the status quo in networking is the introduction of high bandwidth, low error fibre optic communication links in the WAN environment. The cost of providing a fixed amount of bandwidth over WAN links is likely to fall significantly as the major expense is transfered into the laying and maintaining of the fibres over long distances. These costs are roughly the same for a fibre carrying a few kilobits per second as for one supplying a few gigabits per second. Optical fibres also require less signal conditioning and

regeneration than existing copper cables over long distances which reduces the initial capital outlay for very long links.

Fibre optic links are already heavily in use in the voice telecommunications industry. All of the British Telecom trunk network is now using fibre links and many of the cable television companies are using fibre for all new installations and replacements for existing copper installations. The massive bandwidth available on fibre allows these companies to carry more services over a single light guide, resulting in large cost savings. A number of testbeds are currently active worldwide with the aim of using similar technology to deliver cheap, high bandwidth, low error rate communication links to the computer industry. With computer communications making up an ever increasing amount of the telecommunications carriers' revenues and the current political push in many first world nations to deploy "Information Superhighways"[1], these research projects are seen as of vital importance to both the industry and national interests.

It is towards this world of high bandwidth, low error rate, fibre based WANs that the research described in this thesis is targeted. Specifically, work described looks at how these revolutionary changes in long distance link capabilities can be used to efficiently support distributed computations spread between widely dispersed remote hosts. The use of distributed computing in LANs already allows the construction of powerful virtual machines. If that concept can be extended to WANs where high cost, high performance computational engines or machines supporting specialised resources are available in an efficient, cost effective manner, the potential performance benefits are enormous. Work is already underway at some of the gigabit testbeds to make all the machines involved in a distributed computation appear to the user as a single, very powerful system. The resulting Wide Area Distributed Computer Systems (WADCS) appear to be a much needed tool in the computational arsenal of scientists working on "Grand Challenge" problems such as meteorology, biomedical and engineering visualisation and high energy physics.

[1]The author finds the term "Information Superhighway" to often be an overused "hype" phrase and so will refrain from using it as much as possible. However it must be remembered that the political will to push this concept has resulted in the funding of a number of very useful testbeds that would otherwise have been too expensive to develop.

1.2 The Latency Problem

The biggest problem facing WADCS based on gigabit networks is that, while the bandwidth available to an application has risen considerably, the latencies experienced over the links have not fallen by a similar amount. The latency is the amount of time that passes between a packet or cell being transmitted from the originating application to its reception by the target process on the remote machine. There are a number of factors that can affect the latency; the efficiency with which the operating systems of the hosts involved can move packets between a user application and the network hardware, the performance of any store-and-forward switches that the packets or cells must traverse and the fundamental speed of light delay of the signal in the links themselves. The first factors, whilst still present, are making smaller and smaller contributions to the total delay experienced by networked applications thanks to work that has been done of analysing protocol processing in hosts and the introduction of fast switching technologies. However, the fundamental link latency is fixed; it is limited by the speed of light in the link medium and the distance the link traverses. Baring radical break-throughs in physics, the speed of light for a particular medium is an impassable barrier that must be lived with and designed around.

Over LANs, the link latencies are of the same order of magnitude as, and sometimes smaller than, the other factors involved in the total delay. However on WANs link latencies form by far the largest part of the delay experienced between distant hosts. Even with in a single country link latencies can be measured in the tens of milliseconds. Although this does not appear very large in human timescales, it is the equivalent of many millions of machine instructions on modern high performance computers. For such links to be efficiently utilised in distributed computer systems, mechanisms will have to be developed to hide or minimise the negative effects of these link latencies on the overall performance of the system.

1.3 Points Addressed

This thesis describes the in depth investigation of one particular solution to the problem of developing distributed systems over links with high bandwidth-delay products. The mechanism that this research has been based upon is the Late-binding Remote Procedure Call (LbRPC)

system proposed by Craig Partridge. LbRPC is seen as a replacement for the traditional simple Remote Procedure Call (RPC) for use over such WANs. It aims to minimise the number of network transits that a distributed computation must incur in an effort to minimise the effect of the link latency upon the system's performance. It is intended to achieve this by allowing the exportation of arbitrary blocks of code and data to remote machines for processing. This differs from the traditional RPC's exportation of small amounts of data to a fixed range of programs on the remote machines.

Although Partridge's work showed that there was a need to minimise network transits in distributed systems based on high bandwidth WANs, a number of questions remain concerning the actual feasibility of LbRPC. The first of these is whether it is possible to export arbitrary code and data to remote machines for execution. Partridge's work appears to assume that this is relatively trivial, but previous work on universal languages and heterogeneous distributed systems casts some doubt on this assumption. Thus one of the early tasks described in this thesis is a more detailed investigation into this question to determine whether exportable intermediate code and data representations are feasible and, if not, what constraints must be placed on the code and data exported by an LbRPC mechanism.

Secondly, no mention is made in the original work as to how the remote host on which the code is executed is selected from the potentially large number of available hosts in a WAN. This point is of vital interest if the performance of a distributed application that utilises LbRPC is to be even close to optimal. There is little point exporting code and data from a local workstation to a remote supercomputer only to find that the intermediate network links are congested or the supercomputer is already overloaded. Ideally, the code should be executed on the machine that will provide the best overall performance for the application. This thesis presents a number of alternatives as to how an application program may make the process placement decision for code and data exported to remote hosts. The techniques presented range from gathering network performance measurements to a novel use of lower layer multicast communications.

Lastly, the topics of lower layer support and security only received a brief mention in Partridge's thesis. This thesis looks at them in greater depth. Notably, it describes a new transport level protocol that provides support for some of the process placement techniques developed here. The tradeoffs between various potential security facilities and the overall performance of

an LbRPC mechanism are also investigated.

1.4 Outline of Thesis

Chapter 2 provides a more in depth overview of the network environment that this research is intended to operate in, the existing communication mechanisms used to develop distributed applications and the LbRPC mechanism as proposed by Partridge.

The next two chapters look at the issues involved in representing arbitrary sections of code and data respectively for exportation over the network. Each chapter provides a brief overview of existing representations that are available and then investigates how well these mesh with the needs of LbRPC.

There is then a chapter that details the process placement problem and the solutions devised as part of this research to address it. Measurements of network and host performance are reviewed and potential problems are outlined. A number of strategies for performing process placement are then described, culminating in a method that uses multicast network layer communications to distribute the code and data to the remote hosts. A number of these strategies have been implemented and results of tests with the prototypes are given.

Lower layer support is described next, with particular emphasis on the transport layer support required to handle some of the process placement mechanisms described in the preceding chapter. Security issues present in LbRPC are then described along with a number of possible mechanisms to provide a variety of levels of security. This is followed by the conclusions derived from this work and a discussion of how this may then be further developed in the future.

Chapter 2

Survey of Networking and Distributed Systems

2.1 Introduction

Computer networks have traditionally been separated into low bandwidth long haul Wide Area Networks (WANs) and much higher bandwidth Local Area Networks (LANs). This categorisation is mainly a result of the widely differing technologies that have had to be employed for the different networks. The WANs have typically been built around connection oriented protocols running over copper cables with high bit error rates (BERs) of around one error in every 10^4 bits. LANs on the other hand have been based on mainly connectionless protocols running over shared media with a much lower error rate.

A side effect of this categorisation of networks based on geographical coverage is that many distributed applications, operating systems and higher level protocols have only been designed to work well over one type of network. For example, most distributed operating systems described in the literature have initially been implemented on LANs and then gateways have been used to link them together over WANs. Many protocols designed for WANs have been found to have too high an overhead for efficient use on LANs. The most notable exception to this has been the DARPA TCP/IP protocol that is widely used on both LANs and WANs, and that does appear to have much better scaling properties than most other protocol suites.

However, computer networking is experiencing the introduction of a variety of technologies

which are likely to have a revolutionary effect on computer communications and distributed systems. One of the most talked about of these is the introduction of wide area gigabit networks. These are WANs that have the same, or even higher, bandwidth as the LANs that they are connected to. The bandwidth differential between LANs and WANs can therefore no longer be assumed to hold true. As the networks are becoming more and more transparent to both the users and the applications programmers through the use of network and distributed operating systems, it is becoming vital to look at how we can make use of this increased bandwidth to build Wide Area Distributed Computing Systems (WADCS).

In this chapter, some of the revolutionary technologies that underlie the implementation of gigabit LANs and WANs are detailed, along with some of the fundamental problems that have to be faced. This is followed by an overview of some of the testbed high speed networks that have been designed or are currently under construction worldwide that demonstrate that high bandwidth WANs are achievable and will soon be in widespread use. The desire for the development of WADCS is then investigated, along with some examples of systems which are currently running or planned. The mechanisms that are traditionally used to implement distributed computation are then discussed and some of the problems with the existing practices are outlined. Late Binding Remote Procedure Calls (LbRPCs), a protocol designed by Craig Partridge specifically for use over high bandwidth WANs that will be used as the basis of discussion throughout this thesis, is then described and some of the potential problems that are faced in its implementation and usage, that will also be addressed in this thesis, are outlined.

2.2 The Gigabit Revolution

Traditionally networks have been implemented over copper cables and have been the bottleneck in computer communications, mainly because of the vast differences between the high CPU throughputs and slower network. Research into gigabit networks is changing both of these facts. Gigabit networks offer throughputs equaling, or in some cases even higher, than the available memory bandwidth in the host machines. They can also break down some of the barriers between wide and local area networking as they allow high speed communications to take place over large distances.

Gigabit networking is mainly based not on copper cables as a physical medium but fibre optics. Fibre optic light guides offer a potentially huge bandwidth; some estimates have put it as high as 25 terahertz in each of three passbands in each fibre light guide [59]. Fibre optic light guides do not suffer from the electro-magnetic interference that copper cables do and have much better signal degradation characteristics. Thus the error-rate is relatively low, even at very high data rates. Long distance fibre optic links require far fewer repeaters than a corresponding length of copper cabling would.

2.2.1 Protocol Processing

When one starts to switch at the optical frequencies, the throughput of the electronics in the switches and hosts then becomes the limiting factor. The bottleneck in gigabit communications networks has shifted from the network itself to the electronics that form its interface with the computing world. Much research is currently underway to attempt to address these low level issues.

This reversal of the location of the communications bottleneck has some implications on the design of both computer hardware and operating system software. There is some debate in the literature as to whether it is better to perform low level protocol processing , usually at the Transport Layer or below in the International Standards Organisation (ISO) Open Systems Interconnection (OSI) Seven Layer Reference Model [179], in separate protocol processing modules or using the main Central Processing Unit (CPU) of the machine. Those arguing in favour of dedicated protocol processing point out that it allows the host CPU(s) to only be interrupted when actual data packets destined for them have been received and stripped of low level protocol headers and trailers [96]. This will reduce the overhead on the main CPU(s) of the machine. They also point out that the memory bandwidth of many traditional machines, especially of those in the workstation class, is too low to handle the high bandwidth communications coming off the optical networks. This has resulted in the design of some protocols that lend themselves well to implementation in hardware, such as the eXpress Transfer Protocol (XTP) [30, 47, 31] and the transport layer protocol proposed by Netravali et al [117].

Those in favour of continuing to integrate the protocol processing with the other functions of the host on the main CPU(s) point out that separate protocol processing boards lock the

protocol implementations in hardware. This tends to prevent experimentation and refining of the lower level protocols as knowledge about the nature of communications in high bandwidth WANs expands [32]. The CPU to memory channel bandwidth of the host machine is likely to still be a problem for outboard protocol processing engines as it tends to be a restriction imposed by the switching speed of the RAM devices used to implement the host computer's main memory. Clark [32] pointed out that using a 32 bit 10MIPS microprocessor, a reasonable TCP implementation could handle a theoretical maximum transfer speed of 530Mbits/sec but the memory bandwidth with 250ns DRAM chips is only 32Mbits/sec. Although the bandwidth provided by memory subsystems is increasing it is not keeping pace with either CPU or memory subsystem speeds.

Netravali et al [117] also point to the movement of data between the host and outboard processor with its associated operating system overhead as one of the major constraints on the performance of their high speed transport protocol implementation. This is seconded by Partridge [131]:

> [...] the bottleneck will not be in the computation required to implement the protocol, but the cost of moving the packet data into the CPU's cache and the cost of notifying the user process that the data is available.

In the Swedish MultiG system, the protocol is implemented on the Swedish Institute of Computer Science (SICS) Protocol Machine or "6pm". The 6pm is in fact 4 88000 microprocessors from a SICS Data Diffusion Machine (DDM)[136, 58]. Each of these processors has a local cache memory which, through careful use of cache coherency algorithms and the hierarchical structure of the machine, appears as a large virtual global address space. The "host" part of the DDM is thus capable of accessing the "protocol" engine's buffer memory directly without the use of DMA or CPU controlled transfers between them. A similar hierarchically organised multiprocessor outboard protocol processor is envisaged by Netravali et al [117] to support the high speed transport protocol. However here the protocol engine is an entirely separate machine to the host and so data transfers must still take place between the host's main memory and the protocol engine.

The problem of protocol processing performance is possibly made worse if the network

is designed to be a connectionless network service (CLNS) rather than connection oriented network service (CONS) as then each packet must be converted and processed individually. Connection oriented protocols do not suffer as much from this problem as most of the time consuming protocol processing is performed during the connection setup and tear down. The state present during the lifetime of a connection allows processing of subsequent data packets to be performed much more rapidly. However CONS connection setup and teardown times can form an appreciable amount of the total communications time in many distributed processing systems.

Some experiments that have been conducted into the implementation of high performance transport and network level protocols have shown that the layering that is useful in designing protocols can actually be a hindrance in implementing them. This is because a layered implementation tends to result in information being discarded at one layer which would be useful at another. The implementation of the physical layer and the MAC sublayer of the data link layer have been shown to have a critical impact upon the performance of high speed networks [158]. Also implementations of buffer and timer management can be of crucial importance, with a poor design imposing a high operating system overhead in the host [32]. Therefore it is vital to ensure that implementations of protocol stacks are as efficient as possible in high performance networks.

Similar performance problems can occur higher in the seven layer model. For example in the presentation layer there are some very complex protocols that have the potential to adversely affect the operation of the entire system. An example of this is Abstract Syntax Notation 1 (ASN.1)[76, 77]. It has been shown that the Remote Operations Service Element (ROSE) and File Transfer, Access and Management (FTAM) OSI protocols, that both use ASN.1, are consistently slower than their Internet equivalents (SunRPC and ARPA-FTP) on the same hardware [63]. One suggested solution is that it may be possible to implement an optimised form of such protocols in a silicon coprocessor [141]. Of course, this in turn suffers from many of the same problems that affect lower level protocols mentioned above.

2.3 High Speed Networking Testbeds

A number of high speed networking testbeds are currently under development. These are mainly systems that allow concepts to be tested and that prototype hardware and architectures may be evaluated in. However some are also being deployed as full production systems. They will no doubt form the basis of a future gigabit Internet in much the same way that the Arpanet formed the initial research basis for much of the current Internet.

2.3.1 US Testbeds

The United States Government has for some years been encouraging research and development of high performance network technologies. This has resulted in a number of high speed testbeds being deployed throughout the country. Some of these testbeds have been funded centrally by the government through research grants whereas others have been developed privately by the research arms of telecommunications companies.

The US Government's National Research and Education Network (NREN) programme [61] has resulted in the development of six "stage three" high performance testbeds; Aurora, Blanca, CASA, MAGIC, Nectar and VISTAnet. Each of the testbeds is aiming the main thrust of its research at a different aspect of high performance networks research. Aurora and MAGIC are concentrating their hardware efforts at developing Asynchronous Transfer Mode (ATM) technology that will run at gigabit speeds and Aurora is also looking at programming abstractions for use of gigabit WANs. Blanca is investigating the multiplexing of traffic with different service characteristics and CASA is researching the scalability of protocols and processing mechanisms over gigabit WANs. Nectar is attempting to look at how gigabit WANs can be used as backbones connecting high performance LANs together while VISTAnet is investigating the performance analysis of switched protocols.

Each of the NREN testbeds also has a number of demonstration applications that are intended to show how high performance WANs and LANs may be used in scientific and commercial computing. These applications include video and multimedia conferencing, a variety of Computer Supported Cooperative Work (CSCW) applications, biomedical imaging, meteorological simulations, radio astronomy, supercomputer modeling and visualisation and distributed

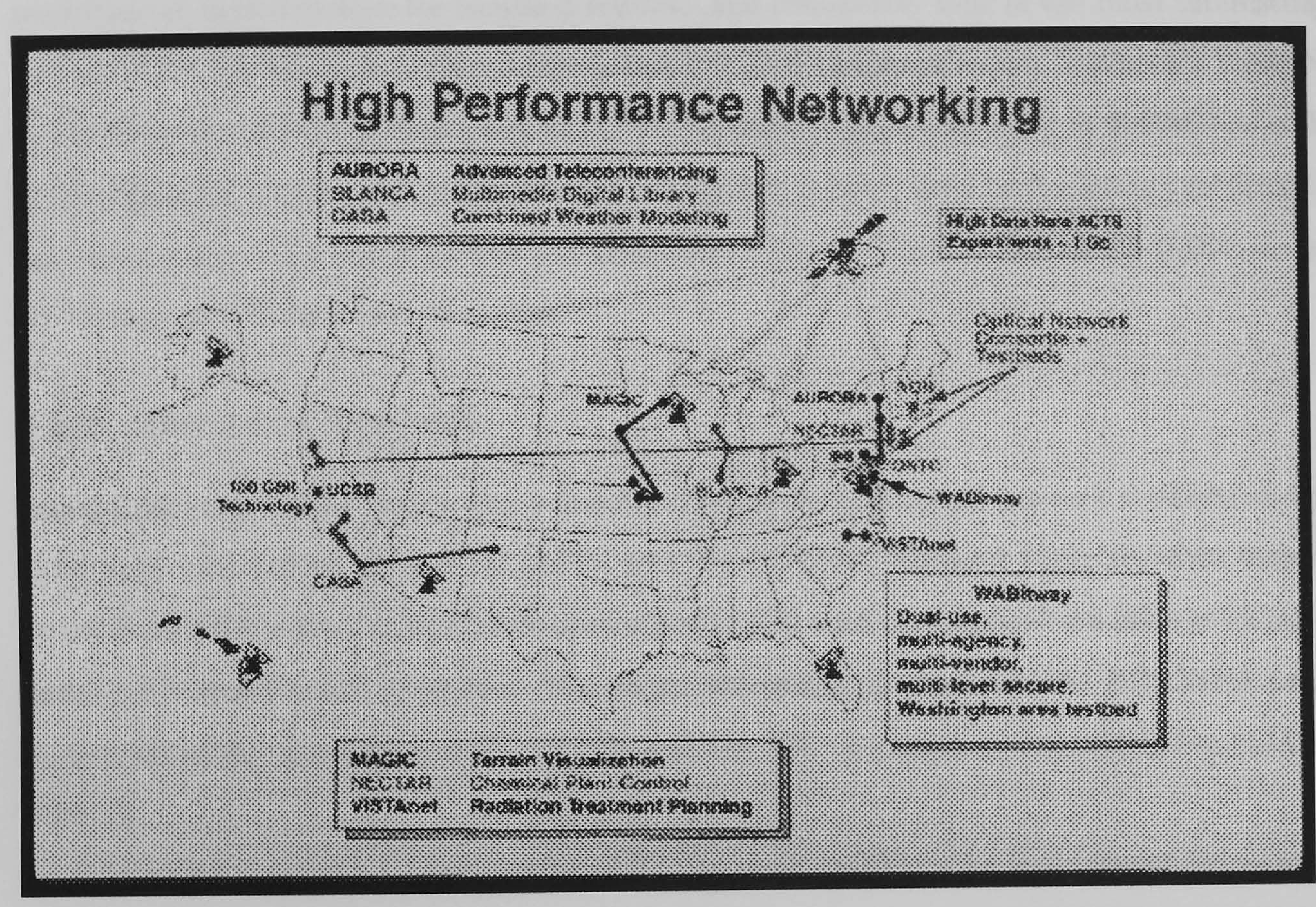


Figure 2-1: US NREN Stage III Testbeds.
Reproduced from the HPCC Blue Book by permission of Joseph B. Evans <evans@tisl.ukans.edu>.

virtual reality.

Other US high performance testbeds include the US Department of Defense's Defense Research and Engineering Network (DREN) initiative [104] that is looking at using "virtual private networks" over commercial ATM services to provide cost effective bandwidth for non-sensitive communications and a variety of regional networks such as the Bay Area Gigabit Network (BAGNet) that are being partially funded by industry and local governments to develop a high performance infrastructure for localised research and commerce. One of the most interesting of the privately funded high performance testbeds is Bell Communication Research's LuckyNet. LuckyNet uses both fibre optics and microwave communications running at 2.488Gbps to link three geographically separated laboratories. LuckyNet is being used to study the problems that may arise when ATM based Broadband Integrated Services Digital Network (B-ISDN) systems are used at speeds in excess of 1Gbps.

2.3.2 European Testbeds

Europe, like the United States, has a number of separate high speed network testbeds and production services either being developed or actually deployed. However these are often targetted at multi-megabit rather than gigabit speeds per se and with an emphasis on cell switching and backbone infrastructure provision. The most interesting of these are the BERKOM project in Germany, the Swedish MultiG project and the UK academic community's SuperJANET network.

BERKOM is a project based on B-ISDN funded by the Deutsche Bundespost Telekom [49]. It was designed to develop and showcase applications running over 140Mbit/s B-ISDN links. The original project started in 1986 (making it one of the earliest high performance networking testbeds) and officially ended in 1992. However its success was such that a separate company was formed in 1993 and the project was given an unlimited future lifespan.

The BERKOM project has involved over 60 partners in both the commercial and research fields. The project has developed a number of broadband based applications including multimedia information systems, computer aided manufacture, video conferencing and electronic publishing. The deployed BERKOM network is fibre optical with 140Mbps links. There were originally two Synchronous Transfer Mode (STM) switches in the network which were aug-

mented by an ATM switch in 1989. Ethernets, Token Rings and FDDI LANs have all been connected to the BERKOM backbone, and BERKOM is now to be used to interconnect ATM based LANs as well.

The MultiG project was a large scale high performance networking research program which has recently been completed in Sweden. It investigated the development and support for multimedia applications in gigabit networks. The project also resulted in the development of a prototype gigabit network based on parallel machines around the Stockholm area. The technology that the network is based upon is a hybrid between ATM and STM called Dynamic synchronous Transfer Mode (DTM). The MultiG gigabit network can be accessed directly from a SICS DDM.

MultiG has looked at a number of collaborative systems during the research project. These have included distributed "white boards", editors and video telephony. The programme has also looked at the tools needed to support the development of distributed applications in a gigabit WAN environment.

The MultiG research program has been completed and a number of interesting results and products have been delivered as a result. The most obvious of these is the gigabit Metropolitan Area Network (MAN) in the Stockholm area, used to link the participants in the project and provide a testbed for developing the applications and support tools. A gigabit wireless network project called WalkStation is now underway to extend the gigabit network to portable computing platforms. The MultiG project also had FDDI and ATM networks installed between its various campuses and these are being used for remote participation in lectures and seminars. A variety of multimedia and CSCW style applications have been developed to run over the MultiG network, including a distributed air traffic control application.

The SuperJANET network is the high speed replacement for the UK academic community's existing Joint Academic NETwork (JANET) network [36, 34]. The network is configured so that many UK Universities, Government research laboratories and similar establishments are able to acquire at least a 10Mbit/s Switched Multimegabit Data Service (SMDS) data links to the SuperJANET IP Service (SJIPS), whilst a smaller number of core sites have a much higher speed service. This high speed service was originally provided by 140Mbit/s Pleisosynchronous Digital Hierarchy (PDH) links that were converted to 155Mbit/s Synchronous Digital Hierarchy

(SDH) links and these may in turn be upgraded to 622Mbit/s links at a later date. The switched network is provided by BT who are providing part of the funding for the deployment of the network and some research projects that are making use of it. Although SuperJANET is not a true gigabit network it is one of the largest high speed networks to form a production part of the Internet. SuperJANET has been used not only as a production IP backbone serving the UK academic community, but has also been used for investigating the use of high speed ATM based network technology for applications such as real time video conferencing and remote visualisation.

2.4 The Need for WADCS

As can be seen from the previous section, high speed, multi-megabit WANs are already being deployed, some as production services. Gigabit WANs are not too far behind, with the constantly growing demand for more bandwidth from the network community ensuring that their research and development will continue a pace. One area that is likely to make heavy use of gigabit WANs, and which is likely to account for the deployment of at least some of the early research systems, is Wide Area Distributed Computing Systems (WADCS).

The desire to build WADCS[1] is to allow many powerful machines to be utilised simultaneously over WAN links to aid in the answering of the so called "Grand Challenge" problems. These problems are from outstanding areas in the sciences and engineering that the US Government has outlined as keys to the opening of new fields for product and service development by industry in the future and to improve the quality of life for ordinary people or society as a whole[2]. The "Grand Challenges" include:

- Mapping and modeling the human genome,
- Increasingly accurate weather and climate modeling[143],
- Energy and environment management[26],
- Space technology,

[1]WADCS are often called *metacomputers* in American literature. In presentations the author has found this term tends to cause confusion and so WADCS will be used throughout this thesis

- Military battlefield simulations.

Although the concept of the "Grand Challenges" was devised in the US, the scale of the problems are such that they are likely to require a world wide effort on the part of the scientists and engineers involved. Part of this collaboration is the requirement to share the processing load of some of the highly complex mathematics and simulations between widely dispersed supercomputers and workstations. Ideally the researchers involved in tackling the "Grand Challenge" problems would like to be able to view the collections of supercomputers available at many different sites world wide as a single computing system. To do so, the communications mechanisms used to build the distributed applications must be able to efficiently support computation over the high speed, wide area networks described above.

2.5 Communication Mechanisms

There are a number of different communications mechanisms and programming paradigms currently in use in distributed computing systems. Here a brief overview of the features of each of the paradigms is given, along with examples of applications and systems where they have been put to use. Also, if one mechanism relies upon another as a lower level building block, a mention will be made of that fact.

2.5.1 Message Passing

Message passing is probably the lowest level communication method, and underlies all of the other programming paradigms. This is the raw face of distributed systems and gives the programmer absolute power over the communications. However this power is acquired at the expense of programming ease; message passing systems require the programmer to understand the underlying communication protocols well and provide little, if any, transparency facilities.

Message passing communication paradigms are heavily used in tightly coupled parallel computers. The low level nature of raw message passing primitives gives the programmer on a parallel processing system the ability to exert a very fine gained control over the parallel threads of execution in their application, which can lead to great performance advantages.

The sockets library, often found in UNIX systems derived from the BSD 4.2 or BSD 4.3 releases [98, 72], provides a set of simple front ends to the UNIX kernel's communication resources. It forces the programmer to select the appropriate transport protocol for their application explicitly and makes the entire connection setup, manipulation and usage highly visible at the application level. It is nevertheless, a widely used communication library and is highly popular for providing IPC for distributed applications.

2.5.2 Traditional Remote Procedure Calls

The traditional Remote Procedure Call (RPC) is a programming paradigm devised by Birrell and Nelson in 1984 [12] that has proved to be very popular amongst the designers of distributed operating systems and applications. In its traditional and most widely used form, RPC is a blocking request and response protocol. The local program calls a stub routine which simply takes the parameters to be passed to the remote procedure and packages them into an intermediate, standardised network format. These are then sent to the remote machine for execution (using a message passing primitive) and the local stub procedure "blocks" pending the return of any result codes. Thus, at least in traditional implementations, RPC is a synchronous protocol. Some projects have extended the paradigm to asynchronous execution by allowing the local program to continue until such time as it attempted to make use of results returned by the RPC call (at which point it has to be blocked).

RPC is highly popular in the distributed systems field, and is a part of practically every research system and all commercial distributed systems. RPC's main advantage is that it is a familiar paradigm for the programmer to use; making procedure calls is something that most application programmers understand very well. However, it does not usually perform well over WANs due to the request-response transactional nature with relatively fixed remote programs. Frequently the size of the data and results is just a few kilobytes at most and the time taken by computations performed on the data can be dwarfed by the latencies experienced in WANs. This is likely to appear more pronounced in future networking environments, where the available bandwidth and CPU speeds are much higher but the latency is fixed. Traditional RPC mechanisms can also place certain limits upon the types of procedures and data structures that can be exported.

2.5.3 Distributed Shared Memory

Shared memory has long been a feature found on timeshare uniprocessor systems and closely coupled multiprocessing systems, and is now finding its way into more loosely coupled systems. The basic idea behind such a system is that processors operating over the network can make use of the same address space for data and possibly even code[2]. Usually Distributed Shared Memory (DSM) systems support multiple readers and a single writer to an area of memory, with the processor that is currently writing to the memory being the "owner". A wide variety of protocols have been devised for maintaining the consistency of the shared memory caches in DSM environments, although most of them are designed solely for use on low latency (if not necessarily high bandwidth) LANs rather than WANs.

Some interesting work has been undertaken on attempting to allow Non-Uniform Memory Access (NUMA) architectures [13, 15] to be built. In these systems, there are a number of different levels of shared memory access time (rather like the normal CPU cache-RAM-disk-tape storage hierarchy seen on many machines today). NUMA systems can work effectively by attempting to increase the localisation of the data which they use a lot using techniques such as processor clustering and memory ownership "ping-pong" [105, 14, 181].

The advantage of using DSM is that, being a very low level form of data sharing, it can be made almost invisible to the application level programmer. It is possible to run the same programs with data being brought from remote memory or from local memory and only be able to tell the difference by minor configuration changes and the increased access times of using DSM. However, there is the potential for the "network aware" programmer to aid the system by giving hints that pages of memory will be required in advance of their use so that they can be requested before they are used and thus already be present on the local machine (therefore hiding the access time of the remote memory access). With the use of tracing and profiling tools, it is possible that this could be performed semi-automatically.

The one over-riding disadvantage of DSM systems is that they make it very hard to support heterogeneous environments. The memory architecture of one host machine is often different to

[2]In the past code has been shared by closely coupled processors but doing doing so in a loosely coupled distributed system is often quite difficult, especially if it is running in a heterogeneous network environment. Even a slight change in the operating system running on otherwise identical hardware platforms can render binary code useless. This same problem is present in process migration mechanisms as will be seen later.

that of another (even from the same manufacturer), with different word lengths, byte orderings and interpretations of data structures in memory. To solve this problem, an intermediate "network memory" model has to be used and that leads to an increase in the overhead of accessing remote memory in the system due to the conversion from network representation to the hosts internal representation (and vice versa in the case of writing).

DSM and NUMA implementations include the Hector distributed research system[184, 180] that has been used as a platform for experimentation with such architectures and the operating systems to support them.

2.5.4 Remote Execution of Programs

Many systems allow remote machines to request the execution of local stored programs. In a large number of UNIX systems this achieved using the `rexd` daemon [68] and the on command [70], which are based upon RPC transactions. These are most often used to allow a user to execute a single command on a remote machine without having to actually log into the remote machine. As the program to be executed must reside in an executable binary form on the remote machine's filesystem, remote execution systems such as `rexd` can be used in heterogeneous environments.

However, they are rarely used in application programs as they provide little location transparency, have a large overhead due to the need to fork off many child shells to run the commands in and do not operate efficiently over WANs because of their reliance upon the underlying RPC protocol. Many implementations also suffer from security problems that are accentuated by the more hostile environment found in large WANs. One niche where such a system has been used successfully is in a modified command shell which performs load balancing across a group of heterogeneous hosts on a LAN [154].

2.5.5 Remote Job Entry

Remote Job Entry (RJE) was (and, to some extent, still is) a method of processing distribution used in batch oriented mainframe environments. It is often simply a case of sending the Job Control Language (JCL) commands and the source code for the program to be run over the network from a dumb job entry terminal to the mainframe. However some protocols, such as

the JANET Red Book Job Transfer and Manipulation Protocol [121], allow peers to send jobs to one another. Unlike remote execution, the executable does not exist on the remote host prior to commencing the communication. Instead, the source code for the task to be performed is sent over the network, compiled on the remote host and then executed.

Problems with RJE are that it often involves a complex JCL itself (sometimes in addition to the JCL required to run the job on the remote machine), is used in a batch oriented, store and forward manner usually and requires a compiler on the remote machine that can handle the specific language being sent (and its specific dialect). Due to this last point, it does not work very well in general in a heterogeneous environment. With the increased availability of high performance workstations allowing easy interactive system usage, RJE protocols are now mainly restricted to high cost mainframe and supercomputer environments.

2.5.6 Process Migration

Process Migration has been another well tried concept in distributed systems research, but has currently failed to make much headway into the commercial distributed systems environments [3, 50, 172, 193]. It is an attractive concept as it allows load balancing to be performed transparently across a distributed system. Process migration mechanisms are usually built upon message passing or RPC based primitives.

However process migration suffers from two major draw backs. The first is that it is somewhat limited to a homogeneous computing environment due to the need to make process images (or code at least) move over the network to execute on a remote processor. The second problem is that some processes rely on specialised hardware or software only available at certain node processors in the distributed system and thus these processes may only run on those nodes, unless the system can accept the overhead of having the migrated process "call-back" to the specially equiped node to make use of its facilities [126, 187].

2.5.7 Distributed Filesystem Access

This is probably the most widely researched, implemented and used distributed computing component to date. A number of network and distributed filesystems have been produced, and some, such as Sun's NFS [71], have become de facto standards in the computing community.

The underlying mechanisms used by distributed filesystems varies greatly however and these can affect their performance and the situations that they are well suited to.

However, there are two basic types of distributed file system; the stateful type and the stateless type. Stateful filesystems are those that maintain information, known as the "state", in the servers as to the files each client open and where the clients are currently positioned within each file. Stateless servers on the other hand merely respond to requests from clients for various services such as reading or writing a block or checking access rights without maintaining any record of the files each client is currently making use of. NFS is a widely deployed example of a stateless file system.

Network or distributed filesystems can be used as a communication method in distributed systems in one of two ways. At the simplest level, files may be used to store and pass temporary intermediate results between two separate processes running in the system. A more subtle approach is that followed by systems which allow named communication interfaces to appear in the filesystem's name space, such as named pipes in some UNIX systems [73]. The distributed applications programs make use of the distributed or networked filesystem to enable them to create and use a communication channel between themselves. As far as the programmer is concerned, the semantics of operations on the named pipes are similar to those of regular file accesses. The transmitted data is not actually stored at all and the underlying communications link may use simple message passing, RPC or other communication mechanisms.

2.5.8 File Transfer Protocols

Although more limited in scope and applicability than true networked or distributed filesystems, file transfer protocols (FTP) can nevertheless be used to implement the sharing of data and code between distributed processors. File transfer protocols tend to rely on message passing or RPC for the control, with data streaming for the bulk transfer of the contents of the files.

Like distributed filesystems, file transfer protocols can be both stateful and stateless. Stateful file transfer protocols were developed first and are available on most networks (FTP on the Internet [138], "Blue Book" Non-interactive FTP on JANET [120] and FTAM [123] in OSI compliant networks). These operate relatively well over long distances as most of the communication time is spent actually bringing data, with very little required for control. However, these

tend to application layer systems and are not often used inside of other application programs for IPC purposes. The interactive file transfer protocols tend to spend relatively large amounts of time waiting for user input compared to actual periods of data transfer. This ties up operating system and network resources unnecessarily.

More recent protocols such as FSP [48], Gopher [1] and the HyperText Transfer Protocol (HTTP) [9] are of the stateless variety. These systems open a connection to the remote host only when there is a need to transfer data and they close the connection as soon as the data has been received. This reduces the load on remote hosts, but in some circumstances leads to transport layer effects becoming evident to users. For example TCP's slow start algorithm appears to interact badly with HTTP transfers for small document sizes as the connection is torn down before the slow start mechanism can fully start to operate [161].

2.6 An Overview Late-Binding RPC

None of the communication mechanisms detailed in the previous section were designed with the needs of long haul, high bandwidth communication links in mind. The changes in the WAN environment which were outlined earlier mean that it may not be ideal to build future distributed systems on top of such mechanisms. Late-binding Remote Procedure Calls (LbRPC) is a distributed programming paradigm devised by Dr. Craig Partridge of BBN Laboratories as part of his Ph.D. thesis "Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment" [131]. It is intended to support the creation of efficient distributed systems operating over the extremely high bandwidth WANs which were detailed previous is sections 2.2 and 2.3.

In order to help reduce the effect of the large latency in WANs, Partridge suggests that the number of network transits made by any one application should be minimised and as much data (and thus work) as possible be transmitted in one go. The thinking behind this goal is that the more work that is performed at the remote end per network transit, the less relative effect the fixed network latency has upon the total execution time of the remote procedure.

In his thesis [131] Partridge introduced Late-Binding RPC (LbRPC) as a variation upon the traditional RPC programming paradigm described in the classic paper by Birrell [12]. The

major addition that LbRPC has over its predecessor is that remote procedures calls contain not only the data to be processed on the remote system, but also the code to implement the procedure.

This code is compiled from an extended form of the programmer's chosen language (for example C, Pascal, LISP or FORTRAN) that allows procedure declarations to be marked as exportable. An extension to the syntax may be required in some source languages to allow the remote procedure calls to be directed towards a specific host. This may at first appear to force the programmer to decide the remote host to use at compile time. However this is not the case as the function that directs a particular procedure call to a host could be implemented to allow the host address itself to be derived from another function. This allows the remote host address to be determined at run-time and gives the potential for some location transparency in application programs.

At compile time, the exportable procedures are compiled into an intermediate, exportable format rather than the binary instruction format of the local host. This gives LbRPC a heterogeneous aspect that it will require in future distributed computing systems, as we can expect a wide range of computing platforms with widely differing capabilities and instruction sets to exist.

The use of a universal, intermediate exportable code representation also means that code can be re-exported to a third party from the remote host if it should need to. This could prove very useful in the situation where an LbRPC query is sent to a remote database on the other side of the world where the indexing and the actual data existed on different remote machines, which may be local to each other. With traditional RPC, one would have to make an RPC call to the remote indexing machine to recover the location of the data and then make another (trans-global) RPC call to the remote data server itself to recover the data. With LbRPC you would send the exportable code to the remote index server which would then make another (local) call to the data server. This would cut out one trans-global call in this example and thus reduce the effect of the long international communications delay.

At first glance, this might appear to simply be process migration by another name. However a number of differences can be found between LbRPC and existing process migration mechanisms:

- LbRPC code is exportable to any platform and thus supports distributed computing in a heterogeneous hardware environment,
- Like traditional RPC it requires no cache management and so works more efficiently over long haul networks,
- It has the same popular and useful semantics which are found in traditional RPC and which have lead to that being included in most major distributed systems research programs and practically all currently available commercial distributed systems,
- LbRPC is intended to be part of the application programming interface and not just a low level kernel communication system.

LbRPC does share many attributes with the REV enhanced RPC mechanism [163, 164] and also with the NICL client-server system [53]. However, LbRPC is intended to be far more general than either of these two systems. Its use of an exportable language means that, unlike REV, it is not necessary to have a compiler for every possible target language on every machine, which makes it a far more manageable system in a large internetwork of homogeneous hosts.

2.7 Potential Problems with LbRPC

Partridge's thesis argues convincingly that attempting to reduce the number of network transits incurred by a distributed computation and maximising the amount of work performed with each transit is a good strategy for use over networks with high bandwidths but long delays. In order to check that this was true for at least one reasonable scenario, one of the first tasks in this research programme was to simulate LbRPC and traditional RPC, and compare their behaviours. The details of those simulations are presented in Appendix 9 and they show that there is at least one situation where LbRPC has better performance than traditional RPC.

However, only a simple hand-coded prototype implementation was developed by Partridge and this leaves a number of questions concerning the feasibility of LbRPC as a practical mechanism for supporting distributed computation over high speed WANs.

The first uncertainty is that Partridge called for a universal representation for the sections of code exported over the network. The simple prototype Partridge developed used hand-coded

LISP routines exported from a C program. However, the number of existing programming languages in use today tends to suggest that there is not a single language that is capable of representing all high level problems adequately. Different programming languages tend to be suited to different application tasks.

The ability to export arbitrary sections of code may also present a problem for LbRPC. It is very easy in many programming languages to develop code that is dependent upon specific resources or architectural features a particular machine. Exporting arbitrary code to remote machines may not be possible for this reason. However, even if it was possible it might not be desirable for performance or security reasons. One of the problems that normal process migration mechanisms have for example is the need to allow migrated processes to make call backs to specific machines to access special resources or interact with the user.

Assuming that it is possible to export at least a constrained set of code and data over the network, the next question that arises is how to choose where to export it to. There is a good chance that there will be a number of remote machines distributed throughout the network that are capable of handling the execution of the exported module. Ideally one would like to export the code and data to the machine that will be able to return the result the quickest. This results in the need to make a placement decision. Such a decision may have to be based on measurements of network and host performance and current computational load. It is worth investigating exactly how this can be achieved.

Once a particular remote server, or set of servers, has been selected to export the LbRPC request to, it is important to look at the support that can be offered by the lower layers of the network software to minimise the number of round trip delays that the request will experience. It is also important to look at how the requests and the responses can be secured against eavesdropping and malicious attacks on the servers can be prevented. WANs, especially those open to a large, public community such as is found on the Internet, offer a much more hostile environment than the LANs that have previously been used to develop distributed applications over.

2.8 Summary

In this section of the thesis, a review of the developments in transmission technology and network protocols and processing that underlie the future gigabit networks has been given. This overview has also drawn attention to some of the possible low level problems that face researchers and engineers trying to design and build gigabit networks. A description of some current high speed networking testbeds has also been given that shows that the technologies discussed are indeed beginning to bear fruit in the form of usable network services. The need for distributed systems that can utilise these links has also been outlined.

Next, currently widely used communications mechanisms have been outlined, followed by a description of why they may not be optimal solutions to communications in future WADCS. To overcome some of these problems, Partridge suggested the LbRPC mechanism. Although LbRPC potentially has many benefits, it also has some problems which have been briefly mentioned. In the following chapters of the thesis, some of these problems with LbRPC are investigated at greater depth and some possible solutions are described.

Chapter 3

Intermediate Code

3.1 Introduction

The major element that differentiates the LbRPC paradigm from that of traditional RPC is the ability to export arbitrary program code to the remote host along with the data to be processed. Traditional RPC mechanisms that only allow the exportation of data to a limited number of predefined routines constrain the application programmer to only exporting certain parts of their application. This is especially true if the remote server is not under the control of the application programmer or the end user. In WADCS it is likely that widely distributed servers will be under the control of different administrative authorities. Some servers may also be serving large communities of users, each having different needs of the server.

The need to export code to remote servers in LbRPC raises a number of problems. Firstly, the representation used to export the code to the remote server must be chosen. This representation must be able to convey sufficient semantic information from the calling application to the remote server to allow the remote server to generate a process to compute the desired result from the supplied data. Choosing such a representation is not as simple as it sounds; the wide choice of programming languages currently available demonstrates that there is no general consensus on such a representation in use today.

If a standard intermediate code representation can not be developed, the next question raised is how many representations must an LbRPC system support in order to cover a sufficiently wide range of facilities to be generally useful to as large a user community as possible. In his

thesis, Partridge rejected the use of multiple exported intermediate code formats:

> "It should be possible to compile virtually any language into the exportable intermediate language, to achieve the goal of having a single intermediate language."

This chapter explores the problems of selecting intermediate code representations by first outlining some possible requirements for the intermediate exportable code format and then looking at the choices that Partridge originally made. It then overviews the use of intermediate code representations in the field of compiler design and software distribution to see if some of the solutions proposed for those situations may be applicable in LbRPC systems. The possibility of selecting multiple code representations instead of a single general representation is then investigated.

3.2 Requirements for Intermediate Exportable Code

The LbRPC mechanism is designed to allow any one of a number of programming languages to be used to develop the distributed application and to also allow the exported code to be sent to any host in a heterogeneous networking environment. From these two conditions it becomes clear that the intermediate code format chosen should be:

- Independent of the underlying hardware and software environments of both the local and remote hosts,
- Independent of the programming language used and have sufficient expressive power to handle programming constructs from practically any source language,
- Relatively painless to implement.

For an intermediate code format to be suitable for use as the exportable code representation in an LbRPC system, it should have a variety of attributes. Firstly, it should hide as much of the underlying hardware architecture as possible. This allows its efficient use in a heterogeneous networking environment, where LbRPC intermediate code may be exported to any one of a number of different hardware platforms, and may even be re-exported from the initial remote host to another. It is also important that the code is capable of being optimised to run efficiently

on the different hardware platforms and is able to take advantage of special hardware facilities where they exist. For example, it is an inefficient use of resources to export LbRPC code to a massively parallel supercomputer if the intermediate code representation enforces a view of the machine as a classical von Neumann sequential uniprocessor.

It should also be possible to make use of predefined libraries of functions on the remote host. These libraries would then be bound at runtime. This reduces the size of the exported code modules, resulting in a consequent saving in bandwidth and network transfer time. Although bandwidth is likely to be plentiful in a gigabit networking environment, there is no need to waste it unjustifiably. Using libraries also ensures that commonly used functions are coded in the native format of the remote host and thus are heavily optimised for speed. Different host architectures can have a great influence over the efficiency of different algorithms as can be seen from the large amounts of effort put into designing new algorithms for specific parallel processing architectures. The exported LbRPC requests could even be cached as precompiled object modules, this would reduce the amount of time needed to run the exported code should it be sent to the same host in the future.

The intermediate code format ideally should not be biased heavily towards any one particular high level problem oriented language, or family of languages. This permits the intermediate code to efficiently encode a wide variety of different languages and also lets the application programmer make use of his chosen languages' special facilities and features without loss of expressive power or efficiency. The programmer should be basing the choice of high level problem oriented languages used in a specific project upon the needs of the problems being tackled, not upon the needs and biases of the intermediate exportable code format used by LbRPC.

It is now possible to look at a variety of alternatives for the intermediate code. Some were suggested Partridge's original thesis while others have actually been used as intermediate representations in other systems. These can then be compared based on the points raised above.

3.3 Partridge's Suggested Representations

The intermediate exportable code representations that Partridge outlined were PostScript, the High-Level Entity Management System (HEMS) binary postfix language, a variety of LISP dialects and the XIL compiler format.

3.3.1 PostScript

The PostScript language [67] is a stack-oriented, postfix language that is widely used for document page descriptions in laser printers and windowing systems. PostScript was proposed as an example exportable code format as it is often used to exchange highly presentation oriented documents between different hardware platforms.

Unfortunately in the real world the platform independence of PostScript fails to work fully and documents can either lose elements such as complex graphics or the entire document fails to be processed. These failures occur for a number of reasons. Firstly the PostScript program that describes the document may have been generated on a platform with vastly different resources to the machine that finally processes the program. This can result in the latter machine running out of physical resources such as memory space. Also the originating platform may have had access to soft resources that the platform performing the end user processing does not have. In the document description application that PostScript is widely used in these soft resource can include font definitions and specialised PostScript dictionaries. PostScript dictionaries are similar to libraries that are commonly used in many systems and demonstrate the need for either self-contained exported code modules or a means of allowing exported code modules to acquire architecture dependent routines from libraries dynamically on the remote server in an LbRPC system.

3.3.2 HEMS

The High-level Entity Management System (HEMS) [129, 130] was intended to allow network management functions to use exported code modules to process queries rather than using simple, predefined request-response protocols. This can be a great advantage when complex queries need to be made in distant devices. These queries may require a very large number of simple

request-response transactions. Each simple request-response transaction would incur a single Round Trip Delay (RTD). Over long paths in a WAN these RTDs can easily dwarf the processing time required to satisfy the request on the remote device. The number of packets that have to been sent to and from the remote device may also be higher for a large number of simple request-response transactions than for a single piece of exported code.

The difference between the binary postfix representation used in HEMS [128, 178] and the requirements for intermediate exportable code representations in LbRPC is that HEMS was designed specifically for expressing solutions to problems in the rather specialised domain of network management, and not for exporting code modules capable of supporting arbitrary applications. As such the language's constructs and expressiveness may not be the most suitable as an LbRPC code representation.

3.3.3 LISP

The prototype implementation that Partridge created to demonstrate the principles of LbRPC used LISP as its intermediate exportable code format. LISP is one of the oldest high level problem oriented languages that is still in common usage. It was initially developed by John McCarthy in 1956 at the Dartmouth Summer Research Project on Artificial Intelligence and then implemented at MIT in the early 1960s [106, 108, 107]. LISP has become one of the leading languages in use by the Artificial Intelligence community. There are a number of different dialects of LISP, with Common LISP [165] being the most popular and nearest to standardisation.

LISP, and derived languages such as NICL [53] and Scheme [160], have been used in a number of systems as a platform independent code format and the basic S-expressions can be interpreted with reasonable speed. LISP also has the advantages of being easy to parse with a very small and simple parser, and having a parenthetical representation which allows extensions to be added to protocols and code in a rapid and upwardly compatible manner. LISP like representations have been put to good use in systems as diverse as the WAIS protocol [88] and the *gcc* Register Transfer Language (RTL) intermediate code format [162].

Like HEMS, LISP was not designed as an intermediate exportable code representation; it was intended to support list processing activities that are common in many Artificial Intelligence

algorithms. LISP builds all of its data structures from lists and atomic cells, and does not employ a strong typing system. Translating between the typing systems of more conventional procedural languages, such as Pascal, and LISP's own data structures would require the addition of extra run-time LISP code to perform the type checking. This would reduce the efficiency of the resulting code as LISP is usually an interpreted language whereas many languages are normally compiled into machine code before execution.

The LISP data structures do give the language one big advantage over other contenders for the role of an intermediate code format. In LISP it is possible to easily support any arbitrary precision arithmetic which is required by the originating language or host. The arbitrary precision arithmetic routines are often supplied with LISP implementations and only impose a relatively small overhead when compared to fixed precision arithmetic in LISP.

3.4 Attempted Universal Languages

The prototype LbRPC implementation used hand coded Common LISP routines exported from C stubs on the local machine[94]. The purpose of the prototype was not to test the performance of the LISP implementation but instead to provide a proof-of-concept of the basic LbRPC mechanism. The prototype did not look deeply in to how the source languages that application programmers use could be efficiently converted into LISP or even whether this is possible in the general case. Unfortunately previous experience in the fields of compiler design and software distribution tends to point to the fact that such general case conversion is extremely difficult. A single intermediate code format representing inputs from multiple high level source languages and producing outputs for a number of binary, platform dependent formats is a panacea that has not yet been fully realised, despite many promising starts as detailed in this section.

3.4.1 Compiler oriented languages

One of the basic precepts of LbRPC, as laid down in Partridge's thesis, is that it should be capable of working with most, if not all, available languages and yet only require the host that supports a LbRPC server to handle a single "intermediate representation". On closer inspection, the need to handle the wide range of existing programming languages in the intermediate

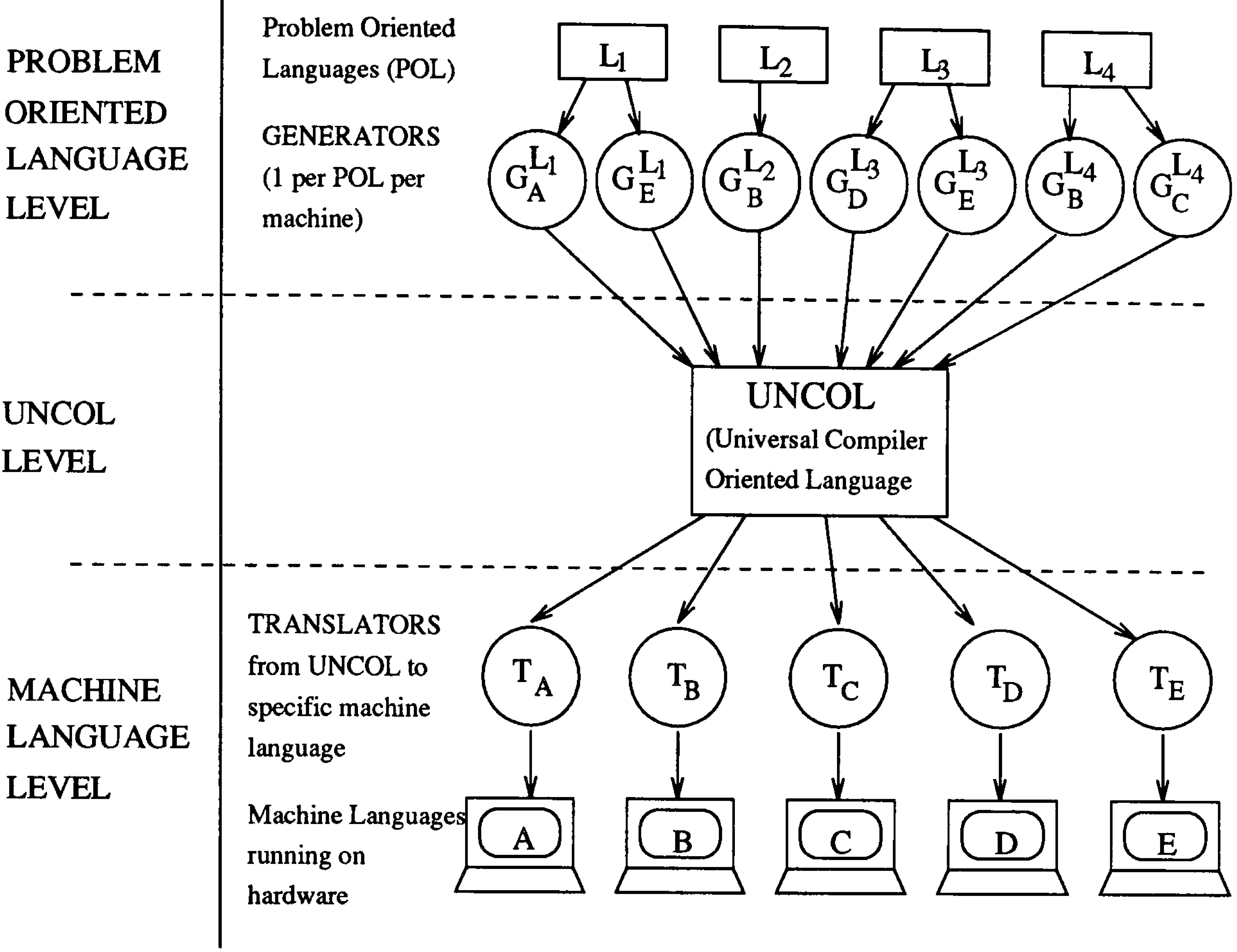


Figure 3-1: The UNCOL 3-Level Concept

representation means that it must really be a "universal language" into which other languages may be efficiently and compactly translated automatically. In this respect it bares more than a passing resemblance to the requirements of the UNCOL proposals in the 1950's [33, 122]. UNCOL was intended to be the UNiform Compiler Oriented Language and was to unify the needs of all programmers into a single compilable language into which other, application oriented high level languages could be compiled. The UNCOL representation of the code could then be used to generate executable binary files on any of the platforms on which it was supported. UNCOL therefore had a "3 Level Concept" of Problem Oriented Languages, the UNCOL representation and the Machine Languages as shown in Figure 3-1.

The initial UNCOL proposals and reports were optimistic as to the effort required to produce such a language. Many of the ideas that were proposed in the original UNCOL effort have subsequently been applied in many areas of computer science including character set and data

format standards. However, the state of the art at the time of the original proposal was sufficiently primitive to prevent much headway being made. For example, basic concepts in programming languages that are well understood today were still undiscovered when the original SHARE report was written [102].

In the next few decades a number of projects were undertaking and proposals produced describing UNCOL style systems [152, 166]. Many of these stumbled either because the intermediate UNCOL representation that they attempted to specify rapidly became large and baroque and were consequently almost impossible to implement, or because they attempted to use a compact, easily implemented UNCOL representation which failed to capture the full range of facilities expressible in the source languages.

The most progress in UNCOL style languages for intermediate compiler representations has occurred in advanced portable compilers in the last decade. These portable compilers utilise a representation midway between the high level constructs of the application oriented source languages and the low level details of a particular hardware architecture's machine code. Examples of this technology include the Portable C Compiler [85], the Amsterdam compiler kit [174] and the Register Transfer Language used in the GNU family of compilers [162]. These representations allow a single basic compiler framework to handle a number of different high level, application oriented input languages and generate executable binaries that will run on a number of hardware platforms. At first such a system would appear to satisfy the platform independence goal of intermediate exported representation required in LbRPC.

Unfortunately, the intermediate compiler representations seem limited in the input high level languages that they can successfully process. Typically, the languages all have similar semantics and control structures with only differing syntax and minor differences in functionality. For example, the GNU compiler family accepts C, Objective-C, C++ and Fortran as input languages. All of the these languages are procedural and share a large number of similar basic control structures. Indeed in the past a number of translation tools have been developed to convert from one of these high level languages to another [183, 167]. This similarity in high level input languages does not guarantee that the systems are capable of efficiently processing high level languages involving radically different programming paradigms such as functional, parallel and logic languages.

Another problem with these systems is that intermediate internal compiler representation is often optimised for the platform on which the binary outputs are intended to be executed[1]. Although a single set of compiler sources can potentially generate a large number of input language to output executable mappings, it is often the case that more than one actual compiler binary must be built to allow cross-compiling between different platforms.

3.4.2 Abstract Machines

It would be fairly simple to separate the front end of the compiler that generates the intermediate code and the backend that converts this code into executable machine instructions and place the back end on a remote LbRPC server. However one could not guarantee that the front end for a particular high level application oriented language would generate the same intermediate representation as the backend on another machine extracted from the same compiler sources but built for a different set of input languages and output architecture[2].

One possible way round this type of hardware dependency in the internal representation is to generate the internal code for an abstract machine, and then export this representation. The remote LbRPC server would then have to convert this representation in a slightly more hardware dependent version that it could then generate a binary executable from. The abstract machine does not need to be tied to a particular hardware architecture and can thus provide facilities to support high level language features that would be too costly or inefficient to provide on real machines.

Abstract machines have been used to implement LISP [4], Pascal [62, page 363] and Prolog[150]. Some research has been conducted into building systems to allow interoperability of abstract machines that support more than one high level problem oriented language. For example the Common Runtime Support (CRS) system [4] supplies a generalised storage and symbol table

[1]In these cases, although the compiler appears to be very portable it is really just a well written application program that makes use of conditional compilation to include the correct intermediate internal representation for the desired input languages and/or supported machine architectures. The GNU compiler appears to be a prime example of this class of compilers.

[2]The GNU compiler's apparent portability and support for multiple languages made its RTL representation appear to be a possible choice for a universal intermediate exportable code representation. However on closer inspection the RTL that the compiler generates is actually very dependent upon the hardware architecture of the target machine architecture and so it is not possible to split the compiler in this way for use in a heterogeneous distributed computing environment.

management system which can support both the LISP Abstract Machine and a subset of the Warren Abstract Machine used in Prolog. It is interesting to note that both the LISP and Prolog source programs are compiled into C in the CRS system however, as the CRS is implemented in C.

The major problem with most abstract machines is that the abstraction is biased towards the needs of a particular high level problem oriented language. A generalised abstract machine architecture based on, for example, a stack with a limited number of operations available is likely to hit upon similar problems to those found with languages such as PostScript. It appears that abstract machines are pitched at too low a level to successfully provide an efficient means of supporting a single intermediate exportable code representation on all hardware platforms.

3.4.3 Universal High Level Languages

Application programmers are supposed to choose the problem oriented language or languages that they implement a system in fairly carefully[3]. The languages used should be those that provide the best mapping between the problem domain of the application and the machine. Different high level application languages have different strengths and weaknesses. For example, Pascal has been used very widely as a teaching language but using the standard version [75] for low level systems work can be awkward as it does not contain the powerful hardware manipulation features found in the standard versions of languages such as C which were designed from scratch for systems programming[4]. Yet C is often considered to be too "dangerous" to expose inexperienced programmers to; it offers many facilities that can easily be abused and make the task of learning programming techniques more difficult than they should be. Different applications require different levels and types of abstraction and it is this ability to easily support multiple programming paradigms that has caused the most trouble for universal languages of all types.

Attempts to produce a single high level, application oriented language that is capable of supplying the flexibility, expressiveness and functionality required by programmers in solving

[3]In reality of course lots of application programmers just use the languages they already know regardless of whether they are the most suitable for the job!

[4]It should be noted however that the Apple Macintosh originally had all Toolbox calls defined as Pascal procedure and function calls and Apple's enhanced version of Pascal was the supported development language for applications on the machine

any programming task have not been terribly successful. The most notable example of this class of languages is PL/1 [97]. PL/1 was intend to be used for the full range of tasks from commercial applications, through scientific computing to systems work. It was actually implemented on a number of platforms and used for developing "real" systems but failed to capture a lasting user base. This was partly due to the difficulty of *fully* implementing the language as opposed to a particular subset and partly because of the difficulty many programmers had in coming to terms with its unwieldy "designed-by-committee" syntax.

Currently, the high level language that appears closest to being completely general purpose may well be C, and its offsprings such as C++ and Objective C. C was originally devised by Brian Kernighan and Dennis Ritchie in 1972 at AT&T Bell Laboratories [41] as a systems programming language for the UNIX^{TM} operating system. After the informal "K&R" standard [92] had been published and widely used, the American National Standards Institute (ANSI) took on the job of standardising C and produced a specification that was published in 1990 [191].

C has one of the widest installed bases of compilers of any existing language, with implementations on machines ranging from supercomputers to embedded systems microcontrollers. The GNU project's freely available highly optimising, portable C compiler, *gcc*[162], provides a means of porting the language relatively easily to new architectures as they appear. The ever increasing commercial take up of the UNIX^{TM} operating system has also helped distribute the language as until recent years, nearly all UNIX^{TM} installations came with a C compiler as part of the standard release. The result of this widespread availability is that C has been used as an intermediate code that a large number of other languages have been compiled into. These include C++ [167], Fortran [183], Gofer [86], Haskell [45], Hermes [100], LISP [4], Modula-2 [114], Modula-3 [42], Pascal [19, 60], Prolog [4], Sather [74], Scheme [7, 170], Simula [83], Standard ML [175] and Web [10].

Due to the fact that C is a widely used systems programming language it is not unexpected to find that it has been widely used in the development of distributed systems in the past. One interesting usage is IBM's *Concert-C* system [5] that tightly integrates remote procedure calls into the C language. It provides the facilities required to create and destroy processes, link them together and then communicate between them directly using language primitives. Concert/C

handles both compile time and run time type checking and is capable of exporting arbitrary C structures between Concert/C processes transparently using existing RPC or asynchronous message passing mechanisms.

However, C does have some deficiencies if used as an intermediate language. Its original design goal of being a systems programming language has meant that its typing system is often not as strong as that of the language which is being compiled into C. This means that it becomes necessary to either add in extra code to perform run-time type checking in the C code or lose some of the security of stronger type checking from the source language. The addition of extra code will increase the size of the program and reduces performance. However, the lack of type checking facilities may allow type mismatch errors to creep into a program during conversion to C. This trade-off must be taken into account and the decision as to whether type checking is performed or not will be based on factors such as the original source language in use and the desired generality and robustness required of the system.

A further consideration is that languages that are radically different from the family of conventional procedural languages, of which C is a member, are more difficult to implement with C as an intermediate code format. The fact that C is quite a low level systems programming language also leads to portability difficulties as different implementations of the C language (especially those prior to the release of the ANSI standard) make different hardware peculiarities visible to the programmer. An example of this is the use of *bitfields* that allow the programmer to handle objects smaller than the usual 8 bit `char` object inside structures. The legal limit to the length of a single bitfield object depends on the natural length of the underlying integer type that it is based upon. This natural integer length varies from machine to machine and is often related the the word length of the underlying hardware architecture. It is all too easy to accidentally produce code which is tied to one particular implementation of the C compiler on one particular hardware platform; exactly the opposite of what is required from an LbRPC system. For example a bitfield that is declared as seventeen bits long is perfectly valid on a machine with a 32 bit architecture but is illegal on a 16 bit machine. Judicious use of C's preprocessor directives can help to increase code portability but there is a limit to the number of alternatives an application programmer is able to specify.

3.4.4 Interface Definition Languages

Another class of languages that have received considerable attention are the Interface Definition Languages (IDLs) that have been used for defining the interfaces between multiple high level languages used in a single system. Many of these are designed to operate in distributed environments. Examples include the Inter-Language Unification (ILU) [82] system's Interface Specification Language (ISL) and the CORBA IDL [37]. The ILU ISL permits descriptions of the interfaces that code modules provide in a large, heterogeneous project environments. These interfaces are the elements of each code module that may be referenced by other modules in the system.

The code modules themselves can be written in a number of application oriented high level languages and can coexist on the same machine or can be distributed over networks. The ISL provides an object oriented, standardised programming interface to these code modules. ISL was designed to allow it to interoperate with existing strongly typed interfaces of traditional RPC mechanisms and other IDLs, such as CORBA's, that only have slight differences in their object models.

The major difference between IDLs and the exported intermediate code representation required in LbRPC is that the IDLs usually don't specify the actual code to be executed on remote hosts in distributed systems. Their task is to merely describe how code modules can interact with one another by specifying the programming interfaces, usually in an object oriented manner. The "real" computation is undertaken by the code in the modules which may well be non-portable and written in different source languages. The IDL ensures that communicating processes are expecting to handle the same number and types of parameters and results. It also ensures that consistent mappings in data formats are generated.

3.5 The ANDF Approach

A final choice for a single intermediate code representation is a version of one of the software distribution formats that are being developed. The most notable of these is the Open Software Foundation (OSF) Architecture Neutral Distribution Format (ANDF)[66]. Standardised distribution formats are designed to allow software manufacturers to produce a single body of code

that can be installed easily on a large number of platforms whilst still allowing the application programmers a choice of high level application oriented languages to work in.

ANDF is the target architecture and language independent code distribution system based on work by the Defense Research Agency (DRA) of the UK Ministry of Defense and adopted by the Open Software Foundation (OSF) [66]. It was originally known as the Ten-15 Distribution Format (TDF) and was created as part of a larger research project intended to investigate the porting of large amounts of software easily into heterogeneous processing environments [149].

Its later take up by the OSF as ANDF is as a result of a perceived need to allow "shrinkwrapped" software to be easily ported between the wide variety of hardware platforms that the OSF-1 operating system is intended to run on. Software vendors can use the ANDF technology to supply a single distribution of their applications that can be installed onto all supported architectures. This is an attractive option as much of the cost of developing and maintaining an application is ensuring that it can be ported to multiple platforms.

3.5.1 Structure of ANDF

ANDF is based upon the concepts of tokenisation [135]. A token is used to provide an abstract interface between bodies of code in an application. The code can only be called upon via its abstract token, which is later replaced by a concrete code definition. This allows type compatibility checking to be performed more easily at ANDF compile time. An ANDF token can be thought of as a macro declarator, used in much the same way as C uses function declarations, that has the ability to abstract a variety of elements in a program [103]. The elements that are abstracted by the ANDF tokens are core primitives that lots of programming languages use [5]. The tokenised form of a program is intended to be independent of the target hardware and as the code modules can only communicate via well defined abstract interfaces, multiple source languages can be used in a single application.

To use ANDF as an intermediate representation, the original source code format of an application must be passed through an ANDF *producer* to generate the tokenised architecture independent code. It can then be passed through an ANDF *optimiser* and different code mod-

[5]Ideally there would be no language specific primitives but whether this is possible remains to be seen when ANDF is used on non-procedural languages such as Prolog

ules can be brought together using an ANDF *linker* to remove as many external references as possible. The result of this process is the program in its distributable ANDF state. The program in this state is then shipped to the end user on a single media, irrespective of the hardware platform that the application is destined to run on[6]. The ANDF distribution code can be sent to an *installer* once the distribution media is mounted on the target platform. The installer combines the application specific, platform independent ANDF code with any application independent, platform specific library functions and generates the architecture dependent executable object file.

Currently, only ANSI C has been actively used to generate portable application distributions using ANDF, using the *tcc* system written by the DRA [149] and an ANDF installation translator based on *gcc* (the GANDF project [55]). The OSF and DRA are still investigating the possibility a multilingual producer based on the GANDF technology.

As ANDF is intended to allow application software authors to program in a variety of high level problem oriented languages, the tokenisation system is being designed to support a large number of possible language constructs and datatypes. The OSF is funding ongoing research looking at using ANDF technology to support languages such as Ada, LISP and Prolog which may require additional primitives to be added to the ANDF system [40]. The goal is to evolve ANDF into a single representation that can be used to distribute any shrink-wrapped code to many hardware platforms.

3.5.2 Pros and Cons of ANDF usage with LbRPC

As ANDF has been adopted by the OSF as a means of allowing software developers to rapidly port their applications between different hardware platforms, it would also appear to be potentially suitable as an intermediate code representation for use in LbRPC systems. Its architecture neutrality, language independence and ability to be optimised and compiled to efficient executable object code are all definite advantages for its use as an intermediate exportable code representation in LbRPC. ANDF has a far greater expressive power for this application than

[6]Unfortunately for the OSF, the economics of distribution have changed since they started working on ANDF. CD-ROM distributions are now very cheap to make and there is an ever growing trend to distribute platform dependent application binaries directly over the Internet which may make the original purpose of ANDF unnecessary.

the languages reviewed earlier in this chapter. It is this wide ranging functionality that gives ANDF the lead over other contenders for an intermediate exportable language.

The ANDF linker and installation system also allows target dependent code to be loaded from library modules held on the remote server, allowing target architecture dependent code to be used in conjunction with an essentially highly portable intermediate code representation. ANDF has a fairly compact representation, with a binary format that results in many of the distributed ANDF encoded programs being roughly the same size as the native binaries. ANDF's textual representation is similar to that of LISP. However, although both of them are tree-structured programming systems, the set of underlying operations which ANDF also offers is somewhat different. Whereas LISP's primitive operations were designed to support efficient list processing, ANDF's primitives are geared towards supporting code representation and distribution. This difference in design philosophy and primitive operations available makes ANDF more attractive as an exportable intermediate code format than LISP.

However, ANDF is not without its disadvantages. Firstly, the process of converting the exported ANDF code to an executable object file can be very time consuming, taking up to several hours for large applications [134]. If ANDF was used as an intermediate exportable code representation in an LbRPC mechanism, the installer would have to be part of the remote LbRPC server and this installation overhead would be incurred everytime code was exported to the remote server. ANDF does not appear to have been designed for possible interpretation of the code by the installer, rather than compilation. Such interpretation may well be possible of course but it has not been demonstrated.

The question of compilation versus interpretation will be investigated in more depth later in Section 3.8. However, this possibly large installation overhead means that it could be possible for the process of translation from ANDF to executable code to take longer than the actual execution of the resulting code. In the OSF's envisaged usage of ANDF, the installation time is not terribly important, as it is intended that the application need only ever be installed once on each new platform. Once installed, the resulting executable application is likely to be used many times and so the installation overhead is spread over many executions. The LbRPC system must factor the installation overhead into the total time that the exported code will take to run. Installation time is far more important in this context as the code is only likely

to be executed once. Even if the code was executed more than once, reinstallation would be required unless the installed code was cached on the remote server. The installation time can therefore make a great different to decisions on where to place code and data at run time as Chapter 5 of this thesis shows.

The next problem with ANDF is that although it is architecture neutral, little work appears to have been done on ensuring its efficiency when used on machines which do not have a classical von Neumann style sequential architecture, e.g. large scale multiprocessors, vector based machines and dataflow architectures. ANDF has been developed so that it is extensible and therefore extra facilities can be easily added should the need arise, but these should ensure that backward compatibility is preserved. This is an example of some of the ongoing further research work for ANDF, and is indeed mentioned in one of the OSF Research Institute's documents [84].

A final problem with ANDF as the basis for an exportable intermediate code is purely a practical rather than theoretical one. ANDF, unlike most of the other languages mentioned previously in this paper, does not yet have a freely available set of code generation and development tools upon which to base experimental LbRPC systems. The time taken to implement an ANDF producer and installer would probably be far greater than the time required to write the rest of the LbRPC system. The OSF's research into using GNU's *gcc* as the basis for ANDF producers and installers is promising, but until very recently no actual software had been made readily available to the research community.

3.6 Avoiding the UNCOL Problem

The difficulties that have been found in designing and implementing universal languages in the past appears to call into question the sense of trying to ensure that LbRPC has a single, completely general exportable intermediate code representation. This is commonly known as the "UNCOL problem" as it is the goal that the original UNCOL initiative started with and failed to solve. It may be instructive to step back for a moment and consider why such a representation is desirable in the LbRPC mechanism.

In section 3.2 the main requirements of an intermediate exportable code representation are

given. The first of these, the requirement that the intermediate exportable code representation has sufficient expressive power to accommodate translations from virtually any application oriented source language, is only required if the LbRPC system is to allow programmers to write their exported modules in their chosen high level language whilst constraining the complexity of the LbRPC protocol and remote servers. There are two ways of avoiding this requirement that will be looked at now; multiple intermediate languages and limiting intermediate code functionality.

3.6.1 Multiple Intermediate Languages

The remote servers could implement multiple intermediate languages. These intermediate representations could then be the same as the source languages that the application programmers would be using to write all of the local application. Alternatively several intermediate representations could provide general support for a particular class of languages such as procedural, logic based or functional. Both of these would allow the programmers to continue to use the programming languages that they are already familiar with and that are appropriate to the problems they are trying to solve. At the same time they reduce the complexity required from the representation actually exported over the network.

However multiple intermediate languages introduce a number of problems of their own. The commonly used high level application oriented programming languages often introduce portability problems due to dependencies on the hardware platforms that they are compiled and executed on, as discussed in the previous sections. This lack of inherent portability means that the application programmer has to ensure that the code that he is attempting to export is capable of executing on all the possible platforms that it might be exported to. The universal intermediate exportable code representation that Partridge proposed seems to have been implicitly capable of rectifying these portability problems. None of the languages covered in previous section of this chapter have that capability.

Another disadvantage to allowing servers to handle more than one intermediate language is that it complicates server implementation and support. Instead of supplying a single installer that interprets the intermediate code representation or compiles it into a binary executable on the remote host, the server now has to include full support for a variety of input languages.

The local client that is exporting the code module also has to ensure that the remote server it is attempting to use can process the particular intermediate language that it is using. This latter problem potentially complicates the process placement decision that the LbRPC local stub may have to make (see Chapter 5) or may lead to undesirable performance if the application is executed in a different network environment to the one that its author was using to develop it.

For example, the author of an LbRPC based distributed application may have access to LbRPC servers supporting languages X, Y and Z. An end user running the same application may only have access to remote LbRPC servers for languages X and Y, so any code modules exported in language Z would fail to find a server to execute on. One way round this would be to tell the user in advance what language servers are required in order to use the application but this then means that the user has to be aware of the facilities available in the network. This breaks the transparency of the network to the end user and that may well not be desirable in many applications.

3.6.2 Limiting Intermediate Code Functionality

The alternative to having remote LbRPC servers implement multiple intermediate exportable code representations is to use a single intermediate representation but do not attempt to make it possible to automatically translate any problem oriented high level source language into it. Such a representation would not have the functionality desirable for handling the general case of exporting any arbitrary piece of code, but should be able to support a reasonable range of tasks that LbRPC is likely to be used for. This single intermediate code format should however satisfy the goal of minimal system dependencies so that it can be safely and easily used on a variety of hardware platforms. A number of choices are possible for such a language.

One possibility is an extensible embedded language such as the Tool Command Language (Tcl) [124] developed by John Ousterhout at the University of California at Berkeley[7]. Tcl is a string based language designed specifically to be embedded in programs written in other languages. It has been used to configure complex application programs, quickly generate window based graphical applications using its "Tk" toolkit and provide a flexible scripting language which makes prototyping services quick and easy. Some have called for Tcl to be considered as

[7]Ousterhout has moved to Sun Microsystems with the aim of further developing Tcl

a possible contender for a universal, multiple platform scripting language as the Tcl core interpreter currently runs under UNIX, Microsoft Windows 3.1, Windows NT and Apple Macintosh System 7. As Tcl's native data type is the string, it can easily implement arbitrary precision arithmetic, although this makes it relatively slow at numeric tasks. The extensibility aspect of Tcl has encouraged a proliferation of extension packages to appear. Two of these packages are especially notable when considering LbRPC mechanisms.

The first of these extensions is a package called Tcl-DP [159]. It provides a set of additional commands in the Tcl interpreter to handle distributed programming demands. These commands include the ability to open UNIX and Internet domain sockets, an RPC mechanism and a simple distributed object system. The RPC mechanism is interesting because the strings that are passed as parameters from the client to the server can themselves contain Tcl code that the server can evaluate. It is in fact a simple remote evaluation system that implements some of the basic functionality of an LbRPC mechanism.

The other interesting extension is known as Safe-Tcl [18] and was originally developed by Nathaniel Borenstein at Bellcore to investigate the potential of active or "enabled" email systems [148]. Active email allows messages to contain not only text, sounds, graphics and video, but also program scripts which can be executed at a variety of points during the mail delivery process. For example, the programs could be executed when the message reached the destination mail transfer agent or it could be delayed until the user actually read the message. Active email allows far more dynamic interactions involving fairly complex processing, such as form filling and validation, group management and synchronised multimedia data streams, to be handled over traditional electronic mail channels. On an abstract level, enabled mail systems can be regard as a variation on the LbRPC. The major differences are that the active messages often do not return a result to the originating host and the use of the mail system itself means they are better suited to applications that run in human timescales, so that network transit times have less effect. LbRPC optimises network usage with a view to the number of CPU cycles each network transit represents; active mail is not so concerned with these details.

Safe-Tcl is interesting as an intermediate exportable code representation as it has been carefully designed with security in mind. Like LbRPC, active email systems offer the ability to have arbitrary modules of code executed on remote machines. System administrators are not

likely to be happy to install such systems unless there are some safe guards in place to prevent accidental or malicious damage occurring to the remote host as a result of executing such code.

Safe-Tcl provides such guarantees by utilising two separate interpreters. One interpreter is trusted and has full access to the Tcl core languages, as well as any extension functions that may have been compiled in. This interpreter is therefore able to delete, rename and create files and utilise system resources, only limited by the permissions available to the user under whose identity the Safe-Tcl process is running. The other interpreter is an untrusted one that has a number of commands in the Tcl core restricted or removed. These commands are those that are capable of altering or removing files or permissions or causing other programs to be loaded and executed. The restricted interpreter also has access to a number of Safe-Tcl extension commands and variables that allow it to find out about the message that delivered it and to request services of the trusted interpreter.

Incoming active email messages are evaluated within the restricted Safe-Tcl interpreter and so code written by an initially unknown originator can not cause harm to the machine. The Safe-Tcl program running in the restricted interpreter can request that certain specialised libraries are loaded and executed for it by the trusted interpreter which allows, for example, extra commands to be added to the restricted interpreter if the email was authenticated as coming from a known, "friendly" user. Commands can be made available to the restricted interpreter by library code running in the trusted, unrestricted interpreter using the Safe-Tcl `declareharmless` command. The `declareharmless` command has to be used sparingly, with care taken to ensure that all the security implications of allowing a new command to be used in the restricted Safe-Tcl interpreter have been considered.

Although Safe-Tcl was originally designed solely to investigate the potential of active, enabled email systems, it has been specified in such a way that the mail system dependent functions, such as processing MIME [17] body parts, are optional elements of the language. This allows the "universal" core of Safe-Tcl to be used in other environments. Some interest has been shown in using it as a safe scripting language in World Wide Web [9] browsers [125] and, coupled with an extended version of Tcl-DP it could form the basis of an embedded LbRPC language with restricted functionality.

3.7 Dynamic Code Generation

One major problem for an single universal intermediate exportable code representation is the need to deal with the ability of languages like Tcl to dynamically generate and evaluate source code at run time. This code may be supplied from external files or remote servers. As such languages can easily generate new source code to be evaluated during execution, when the exported code is running on the remote server, it must be asked how a server that only accepts a single intermediate code format would handle this, and also what the semantics of reading code in from external files should be. It also raises security concerns if the server attempted to "vet" any exported code it received before execution.

To handle the case of the remote server needing to evaluate original source language constructs at run time, the local client stub would appear to have to include a copy of the source language interpreter translated into the intermediate representation. Naturally interpreted languages such as Tcl, Perl [185] and Python [182] provide many subtle ways that code can be introduced at runtime for evaluation and so would require this exported version of the interpreter to be included in all exported code modules generated by the local client stub. Often the interpreted language is only available to user applications on the local machine as a precompiled binary library that they must be linked to and so supplying a version of the interpreter in the intermediate exportable representation might not be possible. If an implementation of the source high-level language interpreter was included in the universal intermediate exported code stream, it would also greatly increase the size of the code being exported and possibly adversely affect the performance of the code being executed at the remote server.

The semantics that should be attached to reading in source code from external files can have two variations. Taking Tcl as an example, the local client stub could read in code from an external file, "inline" it in place of a `source` statement[8] in the exported code and then ship the resulting module to the remote server for evaluation. In this case, the semantics are identical to running the whole application on the local host, as long as the external file is not itself generated by the code that is being exported. However, it is possible that two or more

[8]The `source` statement in TCL instructs the interpreter to read in TCL code from an external file and execute it line-by-line before continuing execution of the original file that the `source` statement appeared in. It is very similar to the C-Shell statement of the same name.

blocks of code could be exported during the lifetime of the local process with an earlier one generating the file on the remote server for the latter to execute. This scenario would require the semantics of the `source` statement to be to read in and evaluate a file of Tcl source code on the host that the code has been exported to.

Of course, two different versions of the `source` command could be supplied to the user that implement both of these semantics. However this solution would then require the application programmer to consider how his code is handled by the LbRPC system which destroys the programming level network transparency. This could be a small price to pay to solve a dangerous ambiguity in the existing semantics. It appears that the application programmer can not be given both a highly functional and flexible programming language and a completely network transparent programming environment in this case. It should be noted that Safe-Tcl side steps this issue by removing the `source` and `exec` commands completely from the initial restricted interpreter. This may also be an acceptable solution in an LbRPC system, although it is another limitation that differentiates exported code from locally executed code.

3.8 Compilation vs. Interpretation

In a LbRPC system the exported intermediate code has to be converted to an executable format "on-the-fly" every time the code is used. There are two choices for this conversion process; the code can be interpreted line by line at runtime, or it can be compiled into a binary executable. It is necessary to look at how slow a conversion process we can tolerate in a LbRPC system. If the code is being exported to make use of some special resources on the remote host that cannot be duplicated locally, such as the use of specialised hardware or licensed software, then longer installation delays are likely to be more tolerable.

However, in the absence of specialised remote resources the goal is to ensure that employing LbRPC is indeed going to be more efficient than simply running the code locally or using an alternative distributed programming mechanism such as traditional RPC or raw sockets to communicate with a remote host. It is after all pointless to try to reduce the number of network transits that a distributed operation has to endure if the alternative takes even longer to accomplish. For LbRPC to provide the optimum means for executing a section of code, the

following inequality must hold:

$$t_{other} > t_{communication} + t_{installation} + t_{LbRPCexec} \qquad (3.1)$$

Note that $t_{communication}$ in the above inequality includes both the inward and outward legs of the LbRPC communications. The left hand side of the inequality can be thought of as the minimum expected execution time for any of the competing methods, such as running it locally or using an alternative communications mechanism. The main problem with this inequality is that all the elements on both sides can only be estimated, and then not terribly accurately. This is compounded by the fact that there may be a number of possible hosts that the code could be exported to, each offering a different performance. Lastly, in many situations, the program under consideration will only be one of many active in the distributed system. The individual behaviour of these programs, and their users, will create an overall system load which will be very difficult to calculate or predict. Thus an answer to inequality 3.1 is unlikely to be computable analytically at runtime. The subject of determining which host to export code and data to will be addressed in chapter 5 but one point to bring out now is the question of whether to compile the code on the remote host or interpret it.

Compilation of code generates a binary machine language program that will usually execute far faster than the if the same source code is interpreted, although if the high level application oriented language has complex semantics with heavy reliance on runtime bindings, the speed can be less noticeable [127]. However, although the executable resulting from compilation runs faster than the interpreting the same code, the actual act of compilation can take an appreciable amount of time. As there is no guarantee that all the code that is generated will actually be used, due to conditional statements and branches, the interpreted code may actually be faster overall.

There are a variety of things that may help increase the probability that inequality 3.1 will hold for an LbRPC server that compiles the exported code modules. For example, routines which have been exported in the past could be cached on the remote host in executable form to allow them to be used quickly again without the need to recompile all the modules; only variables would need to be instantiated with the new exported data. This requires the exported code modules to carry a unique version number that has to be created when the code was compiled

in order to identify different versions of similar code from the same source.

A different strategy would be to interpret the code at run-time rather than compiling, linking and then running an executable. For sections of code that are likely to execute only a limited amount of code, interpretation can easily outperform the compile, link and execute cycle of a compilation based server. Interpreters are also often easier to implement. As Partridge pointed out, the high performance of the computational engines in a future gigabit environment may make the difference between interpretation and compilation negligible when compared to the round trip delays that the exported code modules could experience.

There is little evidence currently available to show whether compiled or interpreted code should be used in a general purpose LbRPC system. Such a decision depends greatly upon the code being exported and the resources available at either end. It may be that both methods should be available and the programmer, or an automated code profiler, be allowed to choose the method to use for each particular piece of exported code. If multiple intermediate code formats are in use, different representations could make different choices regarding the issue of compilation or interpretation, based on the perceived needs of the communities that they serve. In the prototype system developed as part of this research and described in later chapters, an interpreted approach was taken, using the TCL programming language.

3.9 Summary

This chapter has shown that the intermediate code representations put forward in Partridge's thesis have a number of problems associated with them, and the efforts to design universal languages have generally failed to live up to expectations in the past. The most promising avenue at the moment appears to be to limit the functionality of the intermediate language. This means that the programmer will not be able to use arbitrary code in the modules intended for exportation but if sufficiently flexible representations are chosen this may not pose a major problem.

The use of multiple intermediate representations instead of a single universal code format may also be an interesting option. Partridge dismissed this option as it could require a global registry of the servers that implement specific intermediate formats. However as will be seen

in chapter 5's discussion of the placement of code and data, multicasting of exported code modules can offer a solution to this particular problem. Different languages could use different multicast group addresses and as these would be known by the application programmer at compile time, there would be no need to incur any runtime overheads looking for particular servers in a registry. Multiple intermediate representations allow the application programmer to give the same choices of abstractions and features for the intermediate code as are available in problem oriented high level languages.

However it should be made clear that there is no reason to believe that arbitrary sections of code can be exported in a LbRPC system as Partridge originally suggested. There are simply too many opportunities for non-portable code to be written that always behaves differently on different machines. Even the ANDF developers realised this fact and have stated[133]:

> It must be stressed that ANDF is not a "magic bullet". It does not, cannot, and should not mandate portability. [...] You cannot just run a non-portable application through the TDF technology and export it to produce a portable ANDF version.

If non-portable, arbitrary code is seen as a problem for offline software distribution systems, it will surely be a stumbling block for a more time dependent mechanism such as LbRPC. This means that the application programmers must be aware that the code they are writing is destined to be exported over the network for execution. LbRPC may still be able to make the network transparent for the end user but the application programmers must be aware of its presence and the effect it may have on the performance of their programs.

Chapter 4

Exporting Data Structures

4.1 Introduction

The previous chapter looked at how executable code may be represented in LbRPC systems. It is now time to consider how the data that is exported alongside the code may also be represented. The problems involved are similar to that of exporting code; the exported data representation must be flexible enough to support a wide variety of data types, it must minimise dependencies upon both the local client and remote server hardware architectures and it must be feasible to implement.

Like universal languages, the search for an intermediate external representation of data that is manipulated by a distributed system has been the subject of research for some time. However, it appears that this search has been somewhat more successful as systems employing a variety of intermediate data formats are now in wide use in heterogeneous computing environments. Many programming languages have very similar needs when it comes to the data types that they use that helps to constrain this problem somewhat. This is not to say that these external data representations are still not without their limitations.

This chapter starts by looking at how external data representations have been developed and deployed in traditional RPC mechanisms. These mechanisms are probably the most widely used examples of the successful use of intermediate data representations. We then investigate some of the limitations and problems that can occur with these representations and suggest how they may be extended to work in an LbRPC system.

4.2 Existing Representations

A number of external representations of application data have been designed in the past for use in network applications. These representations all fit into the Presentation Layer of the ISO Open Systems Interconnection Basic Reference Model [179] and have one basic goal; ensuring that data leaving one machine is interpreted correctly when it arrives at its destination. Different machines have different native word orderings and data type sizes. Failure to encode exported data in a universally understood and translatable representation can lead to incorrect program behaviour.

4.2.1 Early Work

One of the earliest suggestions for an intermediate external representation for data that was to be exported across a network using RPC, appeared with the initial description of the traditional RPC mechanism [190]. At the time White proposed the RPC mechanism, the needs for external representations of data types was largely unknown. As a result the specification of the data types that could be represented was somewhat vague; it permitted character strings, integers, booleans, "empty" objects, variable length bit strings, as well as lists formed from groups of other objects. It also provided for indexes into lists and character strings. Although floating point representations were mentioned, they were not included in the list of basic data types. This list of data types was intended to grow as more experience was gained with the mechanism.

White proposed two representations of the basic data types; a 36 bit version for use with the then popular 36 bit word based architectures and a "universal" 8 bit binary representation for use between different architectures. The system was intended to make the choice of representation to use at run time using format negotiation in the RPC protocol. The external data representation was specified at the bit level in the words that were to be sent in the packets between hosts.

4.2.2 Courier and Cedar

Work on external data representations in traditional RPC mechanisms improved with Birrell and Nelson's seminal work on the Courier and Cedar [12] RPC mechanisms. One of the five

main requirements that Birrell outlined for an RPC system was the need for strong typing information. It was thus important to have well specified representations of datatypes for exportation across the network if the RPC mechanism was to ensure that valid type checking was performed.

4.2.3 XDR

Currently, one of the most widely used intermediate data formats is Sun Microsystem's eXternal Data Representation (XDR) [69]. XDR was designed for use with Sun's RPC system and also underlies widely deployed services such the Network File System (NFS) [71]. All objects represented using XDR are presented to the network as multiples of four byte words, with each word in big endian order. XDR converts the data in every RPC message carried over the network into a canonical format that include support for a large number of different data types. XDR is actually a language for describing data types passed between different hosts. The range of data types supported by RPC includes:

- 32 bit signed and unsigned integers
- 64 bit signed and unsigned "hyper" integers
- 32 bit IEEE single precision floating point numbers
- 64 bit IEEE double precision floating point numbers
- Fixed and variable length "opaque" untyped data
- ASCII character strings
- Constants
- Void (used for representing requests or results with no data)
- Enumerations (used for representing subsets of the integer space)
- Booleans
- Structures composed of other heterogeneous data types

- Fixed and variable length arrays of homogeneous data types
- Discriminated unions (used for representing values from a specified set of different data types)
- Typedefs (used to create new compound data types from existing data types).

As can be seen from the above list, XDR's basic data types bare a very close relation with the basic data types in the C programming language. This partly belies its growth from the C based UNIX world, where the close match between the two languages made implementing encoding and decoding modules an easier task.

XDR's representation is quite powerful and allows many common data structures used in application programs to be passed between RPC client stubs and the procedures running on the servers. It even allows, through the use of a special case of the union data type known as the "optional data" type, recursive data structures to be represented. However it does suffer from some limitations that will be outlined in the next section.

4.2.4 ASN.1

The OSI network world have introduced the concept of an *abstract syntax* and *transfer syntax* to describe the data types used in machine independent applications [147]. Abstract syntaxes are defined using formal abstract syntax languages and the transfer syntax is used to unambiguously transmit them from host to host. Currently, only one abstract syntax language exists in the OSI protocol suite; Abstract Syntax Notation One (ASN.1) [76]. Two transfer syntax encodings for ASN.1 have been defined; the Basic Encoding Rules (BER) [77] and the Packed Encoding Rules (PER). Like XDR, ASN.1 has seen active service in a number of network systems, including the widely deployed Simple Network Management Protocol (SNMP) [25]. The ASN.1 language covers the data types handled by XDR, as well as some additional ones such as sets and an "any" data type that can represent any other ASN.1 data type.

However, whereas the language used to define XDR is closed, ASN.1 provides a macro facility to allow the ASN.1 grammar to be changed and extended on an application by application basis. Unlike most languages that provide macro facilities, and perform substitutions on the stream of textual input tokens, the macro facility in ASN.1 actually rewrites the grammar rules of the

language. This dynamic rewriting of the grammar rules makes ASN.1 macros an integral part of the language and forces implementations to use a selected known set of macros. Although an interpretive implementation would be able to rewrite its grammar rules, it would not be able to deduce any semantics from the ASN.1 macro syntax and so would not know what structures to generate for an arbitrary ASN.1 macro. This inability to attach semantics to the macros makes a full ASN.1 interpreter practically impossible to implement [147].

The BER was the original transfer syntax for ASN.1 and performs the same task as the network ordered, 32 bit word based format used to transfer XDR representations over the networks. However, the BER does not describe the data being transmitted in low level terms such as 32 bit words but instead represents it as a <tag,length,value> triple. The tag is the data type's abstract syntax and, if several data types are to be exchanged at once, each type is given a unique tag. The length is the number of octets used to encode the value part of the triple. The value encoding itself is the minimum number of octets needed to unambiguously represent the contents of the object with the required precision.

Although the BER is capable of representing the complexity of ASN.1 in a fairly compact encoding, it is not easy to encode and decode it very quickly. This has resulted in comparisons being made between the performance of systems such as the OSI Remote Operations Service Element that use ASN.1 with the BER and Sun RPC with XDR [63]. Invariably, the BER has been consistently slower; sometimes up to twenty times slower. The PER is aimed at reducing this performance gap, but currently ASN.1 seems best suited to applications that do not require very fast encoding and decoding of the external data representation but that do require a very flexible, powerful and standardised mechanism.

4.2.5 Minimalism in Sprite

One last system worth mentioning is the Sprite RPC mechanism [186]. Sprite RPC was designed to support the experimental distributed file system and process migration facilities of the Sprite distributed operating system [126, 187]. As such Sprite uses RPC transactions for a lot of low level network communications between kernels running on different machines. In this application it is vital that the performance of the RPC system was as good as possible. As part of its effort to achieve this goal, Sprite RPC separated the data in the RPC transactions

into two distinct parts for both requests and replies. The first part was the "parameter" space that only contained integer data. The second was an uninterpreted block of data, much like XDR's opaque data type. The integer only first section was found to be sufficient for the simple data structures that the Sprite kernels typically needed to transfer between themselves. The uninterpreted data block was used for character strings, raw file data and for a few special purpose system calls.

Another interesting point to note about Sprite RPC is that unlike Sun's XDR there was no network byte ordering that all hosts were forced to use. Instead the RPC header started with a known four byte sequence. When a host received a packet it would match the received sequence with the known pattern and if they did not match, it would reformat all the words in the packet. Thus in Sprite it is the responsibility of the receiver to ensure that the correct interpretation is made of the data; in XDR both the sender and receiver have to convert to the network byte order. Sprite RPC's Minimalism and high performance are a direct contrast to the generalised, flexible representation and high overheads found in the ASN.1 based RPC mechanisms. However that same minimalism and lack of generality could force the application programmer into adding extra support code to his programmes to overcome Sprite RPC's lack of abstract data types.

4.3 Problems with External Representations

The external representations detailed above have proved themselves to be very effective at supporting distributed systems based upon the traditional RPC mechanism. However, they do have some problems that can limit their functionality and that of the RPC mechanism in general. One of the greatest problems that some, such as XDR, suffer from is the inability to easily represent data structures involving pointers. These are very common in languages such as C and are heavily used in most application programs. C uses pointers heavily, to the extent of using pointers to `char` variables instead of having a dedicated string type.

The problem with pointers is that the runtime system often has little idea what the pointer is actually being used for. For example, to take the case in C of a pointer to a `char` variable, the runtime system may not be in a position to determine whether only the single `char` being

pointed at is important, or whether the pointer to the char actually represents the start of a string. This only becomes apparent when the pointer is actually used and even then it is only possible to really know with knowledge of how the program's function and library calls work. This becomes a problem for external data representations as the system does not know how much data it should export. It could export just the object being pointed to, such as a single char for example, or it could export an entire sequence of objects, such as all the chars from the one being pointed to, up to and including the first NULL[1].

The limitation on the use of pointers in exported data structures can be worked around by the application programmer[2], but it imposes an artificial differentiation between data types that can be used locally and those that are safe to export over the network. As with limitations placed on the type of code that can be exported over the network, this breaks the network transparency that is supposedly offered to the application programmer by the remote procedure abstraction.

LbRPC makes the problem of exporting the data structures required in a operation performed remotely more difficult because it is possible for the exported code module to attempt to reference objects other than formal procedure parameters during its execution. In traditional RPC mechanisms, this is not so much of a problem as the remote procedures have a very well specified programming interface and only require the data structures indicated by the procedure's arguments to be exported. In a block of code exported by an LbRPC system it is possible for references to be made to data structures not passed in as parameters. This is especially true if the program makes use of global data types or the source language used allows called procedures to inherit the local variables of their parent procedures.

In the general case it will not always be possible to determine the actual working set of objects that need to be exported under these conditions. It is necessary in this case to export more data objects than the exported code actually uses. This type of *false sharing* has been found to be a problem in other distributed systems and parallel computers. False sharing increases the bandwidth and time required to send the data to the remote machine.

False sharing can also prevent the code from executing *asynchronously*, with the local client

[1]A C string being represented in memory a sequence of chars terminated by a zero byte

[2]XDR's "optional data" data type can be used to represent recursive structures whose size can be determined at compile time

continuing processing the application code whilst an exported module is being run on a remote server. Asynchronous processing offers the option of introducing some coarse grained parallelism into a distributed system, but as will be explained in Chapter 5 it does this at some cost in implementation complexity and can confuse application programmers who are not used to "thinking in parallel".

4.4 Conclusions

This chapter has outlined the choices in data representations available for use in a LbRPC system. Unlike the intermediate code representations, there are a number of very powerful intermediate representations for data that are already in common use in distributed computing systems. However, there are some limitations with these representations that currently impose some restrictions upon the constructs that programmers can use in their software that is designed to be run over the network.

However, with the difficulty in designing a single exportable code representation as outlined in Chapter 3, the need to be completely general in data representation may not be as vital as it first appears. In reality, the choice of representation used for the exported data will probably be a compromise between compactness, speed of marshalling and flexibility, with different decisions being made for different exportable intermediate code representations. Intermediate code representations that are supporting high level problem oriented languages with many, complex abstract data types may choose to use ASN.1 whereas a more limited representation might use XDR or even Sprite's minimalist approach.

In some cases it might even be possible to consider the code and the data to be represented in the same way. For example, in the TCL based prototype system presented in the next chapter, the code and data are both just strings. TCL uses the string to represent all other datatypes including integers, floating point numbers and even complex list structures. As the TCL program itself is represented as a set of lines that can be thought of as simple strings, it follows that TCL code can easily be used as data to other sections of TCL programs. The idea of interchangability of code and data is not new; it was used in the 1960's with the self-rewriting LISP programs used in early artificial intelligence research. However the idea of making the

code and data representation used in a LbRPC system identical is interesting as it may help reduce the complexity of the remote servers and possibly even enhance their performance if simple enough.

Chapter 5

Placement Decisions

5.1 Introduction

Although LbRPC was designed to operate over WAN links, Partridge did not describe how to go about selecting the machine to place the exported code and data on, given that there is a choice of different machines. The target environment for LbRPC is supposed to be one where there is plentiful, cheap computational power linked by high network bandwidth, but where the networked applications are running over high latency links. As the main goal of LbRPC was to reduce network transits in order to improve total execution time of the distributed application, ideally the LbRPC mechanism should provide the ability to make use of the fastest computation engine out of a set of several. This would allow use of, say, a remote supercomputer rather than a local workstation when it would be advantageous to do so. Unfortunately, it is very difficult to estimate accurately the large number of variables that affect this process placement decision. This is primarily due to the effects caused by the thousands or even millions of other users who are potentially able to utilise the network infrastructure between the local and remote hosts.

A WADCS based on the Internet would contain a very large number of host machines, each with different loading characteristics, resources and network connectivity [111]. The subset of these hosts that a particular distributed application could export code and/or data to is smaller but could still be relatively large (maybe hundreds or thousands of hosts) and potentially widely spread throughout the network topology. This usable subset of hosts will in general not be known to the programmer at compile time and may even vary dynamically during the

execution of the distributed application. This makes determining the best place to locate code and/or data a difficult resource discovery and scheduling problem [189].

In placing a process in a distributed system, there are a number of competing factors to consider:

- optimising total execution time of the distributed application,
- optimising the use of computing resources,
- optimising the use of network resources.

Obviously, in an ideal world all of these optimizations would be achieved. Unfortunately it appears that in the real world this is a very difficult problem [3]. Attempting to optimise one factor often has an adverse affect on the others. For example, trying to optimise the use of network resources by ensuring that code only executes remotely if it will be faster than executing it locally, estimates of how fast the code will execute on various hosts will need to be made. These estimates may well take a lot of CPU time to make, meaning the computational usage and total execution time are no longer optimal. Also, due to that fact that many of the variables involved are difficult to accurately determine, one is still not able to guarantee that network usage will be optimised.

From the point of view of a user of a distributed application, the factor that is most apparent to them is the total execution time of the whole application [132]. We are moving to an environment where network bandwidth and computational resource are becoming relatively cheap commodities but where end-to-end communications latencies are fixed and relatively large. It may therefore make sense at the moment to attempt to optimise just the total execution time of the distributed application possibly at the expense of a little wasted network bandwidth and/or CPU time.

When building a distributed system, the performance of the underlying network can be of vital importance. Before starting to make placement decisions for the objects in a distributed system, it would seem desirable to know, or be able to estimate, how the network connecting the objects will perform. Unfortunately, this chapter will explain how the large number of independent influences present in wide area internets conspire to make this problem almost intractable.

As an LbRPC system must be able to place its code and data for execution somewhere in the network, we then look at the possibility of using multicast communication groups to allow the export code and data to a large group of hosts. This raises the possibility of providing a low execution overhead and so approaches the optimal total execution time for the code and data. It does this at the cost of some "wasted" network bandwidth and CPU time and so is only really applicable for use in situations where these resources are plentiful (as they are expected to be in future gigabit WANs). The rapid spread of the global MBONE [44, 52] virtual multicasting network overlaid on the physical Internet and the fact that future versions of the IP protocol are being designed with multicast support[21] in from the outset suggests that this is a mechanism that will also be widely available in future high performance networks.

This chapter starts by looking at some of the existing work that has been done in network performance measurement. It then looks at the work that has been performed on load balancing and process placement in tightly coupled parallel computing systems and how that may be related to process placement over WANs. Next, a prototype mechanism for determining network performance characteristics is presented using existing, widely deployed network management tools. This is shown to be less than ideal and exposes a number of problems with collecting these types of statistics for making process placement decisions. Then, multicast communications is reviewed and details of a prototype implementation of a multicast process placement concept and its performance are given. This is followed by a discussion of the implications of using multicast communications to support LbRPC.

5.2 Network Performance Measurement

Some research has been conducted on determining the performance of WANs, but little of this is directly targeted at the needs of process placement in WADCS. Instead, many of the teletraffic studies and network models have been devised for either network management and capacity planning or the provision of real time multimedia over packet-based integrated services networks. In the network management and capacity planning case, one can usually afford to expend considerable resources in accurately capturing performance data or running simulations to model the network. In distributed computing systems, network performance is merely one

facet of the process placement issue and would ideally be measured or estimated with as little resource usage as possible. After all, there is little point in devising a distributed computer system that has to utilise most of its network resources in determining the network performance characteristics. This would be a classic case of the act of measurement affecting the variables being measured.

Real time multimedia communication systems also have a need to minimise resource usage whilst still ensuring that the network is capable of delivering the required performance. The difference between real time multimedia and process placement communication patterns is that multimedia communications streams tend to be much longer lived than process placement ones. A four or five hundred kilobyte program might require a fraction of a second to send across a high speed WAN whereas a networked video and audio stream might exist for minutes, hours or even days and carry many hundreds of megabytes of data. In a real time multimedia environment one is ultimately dealing with meeting human time scales so that the users are satisfied with the response of the system. In an LbRPC mechanism, one is dealing with interprocess communications with correspondingly more demanding time constraints. Thus some of the mechanisms devised for use in multimedia communications are unlikely to work effectively over the much shorter time scales required by process placement.

5.3 Process Placement in Tightly Coupled Systems

Process placement in parallel machines typically models network performance as a simple measure of time that communications take. This is a constant in simple models where there is a fixed communications overhead or a value from a mathematical distribution, such as a Poisson distribution, in more complex models where the communication paths are shared between different nodes. In either case the communications time is often very much lower than the time taken to actually execute the code in the parallel programs and is often assumed to be just a few microseconds.

Although there has been quite a lot of work on process placement in tightly coupled distributed systems [119, 3, 194, 151, 24], there has been far less work done for WADCS [121]. This is probably a result of the bias of much past distributed systems research into producing

systems that worked efficiently in LANs but that did not scale to large WANs or only added WADCS support as an afterthought.

The need to decide where to place processes stems from the fact that there are often a large number of potential target hosts in a distributed computing system that could be offered the code and data exported from a process. Process placement is often concerned with load balancing between the various processors in order to ensure that the demands of the various distributed applications in a system are spread fairly over all the processors. It is therefore often used to attempt to optimise the usage of computational resources in a distributed system.

This load balancing is either *static* or *dynamic* in nature. Static load balancing mechanisms make placement decisions once (either at compile time or when the program is first executed) and thereafter make no change to the placement of processes in the system. This means that if the loading in the system changes over time, the load balancing mechanism will only come into play when new processes are initiated and will still not be able to reduce the loads on hosts that already have too many processes. Static placement also suffers the NP-completeness of general optimal scheduling and the lack of good methods for estimating the time a process will take to execute and communication delays it will experience [155].

Dynamic load balancing techniques on the other hand can move processes from processor to processor in response to variations in the load experienced. This means that the system has to have both a *location policy* to state where loads can be transferred to, and a *transfer policy* to determine when transfers should occur. Both of these make use of *threshold* values that represent the current load on specific hosts. The hosts in a distributed system can also perform the dynamic load balancing cooperatively or non-cooperatively by either considering the load on other hosts in the network or by only concentrating on the load on the local host.

Non-cooperative transfer policies run the risk of overloading a host that already has a large working set of processes, and thus this load balancing environment can lead to a process "ping-ponging" between hosts as they continually try to off load it. Non-cooperative load balancing mechanisms tend to be sender initiated. Receiver initiation more commonly found in a cooperative load balancing system.

All load balancing mechanisms must implement an "information policy" that gives the method for exchanging host loading information. This policy can be either *state-dependent*

(also known as "deterministic") or *probabilistic* (sometimes called "non-deterministic"). In the former, decision making is based upon the current state of the whole system and, in the latter, upon sets of probabilities as to what each host estimates the loads on other hosts will be. State-dependent information policies require network usage to communicate the current load status between hosts that may mean that load balancing mechanisms built around this type of policy may not scale well into WADCS unless a careful check is made on the number of hosts engaging in information exchange, the period between exchanges and the size of the data shipped.

A load balancing mechanism must also have a *control policy* that can either be *centralised* or *decentralised* and determines how the host loading data is collected. In the centralised case there is a single central host in the network that gather loading information about other hosts and that makes load balancing decisions for all the hosts in the system. This is relatively easy in tightly coupled or small LAN based distributed systems [192], but is much harder in WADCS where there are far more hosts and network connectivity to the centralised decision making host can not be guaranteed. In these cases, distributed control policies need to be employed. The problem that then occurs is that the local host making its own load balancing decisions is unable to gain a global view of the state of loading of the system as a whole.

5.4 Optimising Network Usage

Efforts to optimise network usage in the placement of code and data have traditionally grown out of the desire to avoid transit latency and congestion. There has also been some demand to minimise WAN usage due to the cost of dial up and per packet charging schemes used in some computer networks. Whether this will be important in the Internet remains to be seen. Currently, many Internet users are accessing the remote resources over fixed cost leased lines from educational, research and commercial institutes. Thus the cost of the link is the same whether they send one packet or one million. In the future a much larger fraction of the total number of Internet users will be using dial up or accessing the Internet via cable TV facilities. They may end up paying per packet and so will wish to minimise the usage of the network in order to keep communication costs down to a minimum.

In the past some distributed applications have optimised their network usage by judicious

use of localised caches. Examples of this can be seen in various network filesystems, especially those designed to work efficiently over WAN links such as AFS. Information retrieval systems such as the FTP [138], Gopher [1] and the World Wide Web (WWW) Hyper Text Transfer Protocol (HTTP) [9] can also take advantage of caching to improve performance [20].

Unfortunately, caching does not work well with dynamic objects such as WWW Common Gateway Interface (CGI) scripts [116]. CGI scripts act in a similar way to traditional Remote Procedure Calls (RPC) [12]. To initiate a CGI script, the user's WWW browser sends an Uniform Resource Locator (URL) [8] containing `<name, value>` pairs that are passed to the script as environment variables. After the script has executed, the results are returned as a document that the browser application interprets using MIME types [16]. As the returned document is highly likely to be different for different `<name,value>` pairs and may even be different from invocation to invocation with the same pair, any cached copy is unlikely to match the results that would be returned from the remote server.

The centralised load balancing policies devised for tightly coupled systems mentioned above will simply fail to work due to the scale of the network and the latency that can be experienced crossing it. No single host in the Internet can be aware of the state of the entire network, or even a sizable set of widely dispersed hosts, at any given point in time. With round trip times measured in hundreds of milliseconds, loading data received from a distant machine can be out of date before it arrives on a probing machine. Thus if load balancing is to be considered at all in an Internet based WADCS, it must be distributed and it should allow potential processing hosts to either initiate the transfer of jobs, or ignore or reject jobs submitted to them from other machines.

5.5 WAN Performance Measures

In a WADCS, the network interconnecting hosts that a process may be placed on is very much larger and more complex than in a tightly coupled parallel machine. The number of CPU nodes available to handle a process is also potentially much larger. The links interconnecting the various nodes may have widely differing performances and may even be constructed of a number of "hops" over physical links that themselves have different performances. Take for example,

the case where one node is connected to another by a lightly loaded 10Mbit/s Ethernet and compare it to the situation where two nodes are on separate LANs connected by a large number of hops using busy Internet links shared with a variety of other traffic. In the first case, it is likely that we can make a reasonable prediction of the communications overhead in distributing the processes between the two nodes. The second scenario makes reliable prediction much more difficult because there are far more unknown influences that can affect the communications.

One major problem of determining network performance characteristics in WANs is that most methods rely on being able to determine or estimate the performance of each of the physical links that communication paths for the process placement will make use of. This is not an easy task, not least because many WANs are packet switched networks with multiple redundant links. This means that two nodes may actually make use of a number of physical paths between the nodes to support the transmission of a single data stream.

Although it is common in the literature to talk of the bandwidth-delay product of a link or hop, this is not an easy thing to measure as it is constantly changing. One only has to run the UNIX `ping` command [169] to a distant host over the Internet during a busy period of the day to see how the latency seems to vary. However, `ping` tells us very little; it tells us is the total run-trip time, *including processing time on the remote host.* Thus a heavily loaded host being accessed across a lightly loaded link can give the same `ping` results as a lightly loaded host access via a heavily loaded link. The `ping` command also tells us little about the bandwidth of the link and can not make any guarantees that it uses the same path through a packet switched internet each time.

5.6 Dynamic Behaviour

One problem with attempting to measure the throughput available over a path in a system such as the Internet is that this is very much a moving target. The throughput between two hosts separated by WAN links in the Internet can vary from second to second and is influenced by a large number of parameters.

The first of these influences is the route that the packets take from one host to the other. In an IP based network such as the Internet, the routing topology is in a constant state of flux

[22]. The variety of routing protocols used in the network will dynamically re-route traffic down different paths to avoid broken links, congestion and "expensive" links. Thus not all packets in a TCP stream may follow the same path through the Internet. The different routes taken may well have differing raw bandwidths. For example, the first half of a TCP stream may go along a path that takes it over a high bandwidth leased line but if that line fails, the remainder of the stream may traverse a lower bandwidth back up route.

Another major factor affecting the throughput delivered from a data stream to an application is the congestion from other traffic experienced at different points along its path through the network. Even if the route followed by all packets in stream is the same, each packet is likely to compete with a different number of packets from other streams for resources at each hop. The burstiness of much of the communications traffic in a computer network makes this situation even worse. If a sudden burst of activity from a number of hosts whose paths coincide occurs, there is a good chance that some of the packets from some of the streams will be lost due to buffer overflows.

An increase in the delay experienced by a packet will also reduce the throughput of a link, as there will be a sudden jump in the packet inter-arrival time in the middle of a stream [93]. The delay increase could be caused by the routing changes or congestion mentioned above or just by slow processing by some device along the path of the packet (for example a host that is also acting as an IP router will have its packet processing rate reduced if a large, CPU intensive job is suddenly started).

All of these factors conspire together to make it a difficult task to estimate the throughput an application's communications streams can expect to experience. The best that can be hope for is to get a "ball-park" figure that is within some bounds unless the bandwidth required is specifically reserved along the entire path through an Internet. This use of resource reservation has already been proposed for real-time connection-oriented communications streams that are found in multimedia communication systems.

However it suffers from a number of problems when used in an interprocess rather than person-to-person environment. The most obvious is that the connection setup time is small compared to the average lifetime of a person-to-person communication session but can be quite large when compared to the average interprocess communication stream lifetime in a distributed

system. Many computer communications mechanisms are based on connectionless datagram protocols such as UDP and these are do not have a connection setup phase. This makes resource reservation much more difficult. Also all the routers along the path must support the resource reservation protocol in use. If just one router does not support it, the guarantees that the protocol can give are effectively null and void.

All of these point to the need for an architecture for making throughput estimates in the current Internet if an even vaguely accurate explicit process placement decision is to be successfully made. Any measurements required to make these estimates should impose a relatively low load on the Internet infrastructure so that they do not interfere too much with other traffic. It must be recognised in advance that they will not give the applications any guarantees of the effective throughput; merely estimates that hint at the network performance that the applications can expect.

5.7 Strawman Proposal

The architecture for estimating effective application to application throughputs described here makes use of existing tools that are already deployed in the Internet. The rationale behind this is much the same as that of the Netfind [153] service; it may be that much of the information we require is already available in the network but we must find it, gather it together and process it into a useful form.

A "straw man" proposal for making the initial estimate of the throughput of a path through the network is to say that it will be the throughput that the sending host experiences on its local network. Due to the low aggregate traffic rates on most LANs, this figure is likely to be around 50-80% of the raw bandwidth of the LAN that the sender is attached to. This proposal has the advantage of not requiring any real use of the network; one can just look at what the network interface on the host has seen in the past. The resulting estimate is likely to be rather poor for communication to hosts that are not on the same LAN as the sender. It takes no account of the variabilities of raw link bandwidths and traffic congestion on other hops on the path through an internet as described in the previous section.

However, it does give us a maximum bound; we know that no path from the sender to any

remote host on the Internet can have a throughput that is better than this figure. As this estimate has such a low overhead for acquisition, it may be a useful "sanity check" to ensure that the local networking infrastructure has sufficient bandwidth available for an application to usefully export a section of code before we start to worry about the bandwidth of each the hops in a path to a remote host over the Internet.

5.7.1 SNMP and ICMP Probing

The next step is to attempt to determine the raw bandwidth of between the nodes at every hop in the path across the Internet. The most obvious solution to this problem is to make use of the Simple Network Management Protocol (SNMP) [25] Management Information Bases (MIBs) [109] that are present in most Internet routers and many hosts. The wide deployment of SNMP agents in Internet routers means that no new software must be deployed in all the routers in the Internet for reasonable estimates of throughput to be made.

The standard MIB described in RFC1066 includes a number of object groups and objects within those groups that can be made use of in throughput estimation. The objects and groups in the MIB are defined in the RFC using Abstract Syntax Notation 1 (ASN.1) [76, 77]. The first of these of interest is the object group "`interfaces`". This group contains a number of objects and sequences of objects that describe some of the attributes of the physical interfaces on a remote machine (be it host or router) that are capable of sending and receiving IP packets.

In the `interfaces` group the object `interfaces.ifTable` holds most of the important information. It is a sequence of `ifEntry` objects, each describing a particular physical interface. One important entry to note for throughput estimation is:
`interfaces.ifTable.ifEntry.ifSpeed`
that RFC1066 describes as:

> "An estimate of the interface's current bandwidth in bits per second. For interfaces that do not vary in bandwidth or for those where no accurate estimation can be made, this object should contain the nominal bandwidth."

This object is mandatory and so all SNMP agents that claim to implement the standard MIB must be able to process requests for this object. Note that although the RFC states that

the

`interfaces.ifTable.ifEntry.ifSpeed`

should try to give an estimate of the current bandwidth on the interface (that is to say the effective bandwidth available to all streams passing through that interface), most routers appear to give the raw link (or "nominal") bandwidth. Thus a physical interface connected to an Ethernet would return a result of 10000000 despite the fact that the CSMA/CD method used to gain access to the medium means that it is impossible for an application to achieve a throughput anywhere close to this.

For IP routers, the standard MIB also defines the objects:

`ip.ipRouteTable.ipRouteEntry.ipRouteIfIndex`

and

`ip.ipRouteTable.ipRouteEntry.ipRouteMask`.

The first of these objects maps the routes to specified IP subnets to physical interfaces on the router, whilst the latter gives the netmasks that are logically ANDed to the destination address of a packet to give the destination network number.

These three MIB objects can be used in conjunction with ICMP Echo requests to determine the maximum bounding bandwidth along a path through the Internet. The basic idea is to send an ICMP Echo Request with an initial Time-To-Live (TTL) of 1. This will get as far as the first router before timing out, causing the router to return an ICMP Time Exceeded response to the sending host. The host can then repeat the operation with a TTL of 2 to find the next host, and so on until the remote host is reached and it responds with a Echo Response. This is the method used by Van Jacobson's `traceroute` program[79], that is now a commonly used network measurement and management tool.

The resulting list of IP addresses from a traceroute output can then be used to make SNMP queries to the routers along the path in an attempt to determine the maximum raw bandwidth. First one would query the `ipRouteIfIndex` and `ipRouteMask` entries to determine the physical port packets are being sent to from the previous and/or next IP address listed in the traceroute output. Once the port[1] has been found, an SNMP query to the `ifSpeed` object can be made to determine the raw bandwidth of the link connected to that physical port. This process is

[1]Note that this is a physical network interface port on the router, not a software based IP port.

repeated for every router in the path traceroute returned.

5.7.2 Problems

The most obvious problem with the above system is the paths can easily be encountered where either the routers are not equipped with SNMP agents or, more usually, the SNMP agents do not allow queries to be made from any random host on the Internet. The reason for the latter restriction is mainly due to the fact that SNMP agents can be instructed to change values as well as reading them. Allowing any Internet user to remotely change important configuration details of routers and hosts is obviously asking for trouble.

One way to help limit this problem is to be able to use the results from routers that do answer SNMP queries to determine the nominal bandwidths of the links to both the previous and next nodes in the path. Using this information, two answering routers either side of a non-answering router will provide enough information to determine the minimum nominal link bandwidth over the two hop path between them.

A more fundamental problem is that this process is slow and resource intensive. A six hop path that takes traceroute less than a second to deduce can take a simple minded SNMP probe several minutes to make an estimate of the raw link bandwidth. If the path is transoceanic or transcontinental with very long RTDs, then this figure can be very much worse. This is because the SNMP query system using an RPC style request-response protocol that is badly affected by high RTDs, and also some of the MIB objects that have to be returned from core backbone routers can be quite large.

The fact that it is slow and resource intensive means that this method should only be used when there is no other data available. For example, if an application program knows that it may contact a certain remote host soon that it has never spoken to before, it may fork off a process or contact a system server to attempt to retrieve the bandwidth estimate for that host while it is doing something else. However, it is not wise to perform the bandwidth estimate more than once in a certain period. How long that period is depends on the stability of the routing in the network and the amount of communication traffic between the local and remote hosts. As a traceroute is fairly quick compared to the SNMP queries, it might be possible to use it periodically to at least check that the path packets are being routed down is the same.

A similar tradeoff between bandwidth usage and accuracy has had to be made by the Internet Engineering Task Force (IETF) Domain Name Service (DNS) Working Group [23]. In attempting to deal with name resolution "load balancing" in clusters with a single DNS name, they have to make a compromise between the bandwidth and CPU time used by frequent zone updates with need for very low refresh rates in the secondary DNS servers if the information is to be useful. They too noted the need to have some intelligence in the program that indicated that a zone refresh was required so that they could avoid unnecessary periodic zone reloads.

5.7.3 Using Heuristics

One way to help alleviate the problems due to resource overloading is to only use the previous SNMP based method for determining the initial bandwidth estimate and then use heuristics based on actual observed throughputs from communications streams from the local machine to remote hosts. This will either require the applications in the local host to make their own observations of the throughput, or modifications can be made to the operating system packet processing software so that it can monitor throughput.

The former is likely to be the better of the two for a number of reasons:

- It does not require modifications to the host operating system software. This is important as much work has been done in optimizing the network processing code in hosts and this will become increasingly important as the electro-optical bottleneck becomes more apparent.

- The application is the only piece of software that is likely to be able to make an accurate estimate of the throughput as it will have to have knowledge of the expected processing delays at the remote end of the communications stream, or the cooperation of the remote host in adding time stamps to its responses. The operating system could only measure throughput over quanta of time and could not be expected to take into account application specific delays.

- It is in-keeping with the "micro-kernel" approach to operating systems design that attempts to devolve as much processing as possible to user code.

The disadvantage of letting the application monitor the throughput on its own is that multiple applications talking to similar sets of remote hosts will be recording similar information. Unless there is some central service on the local host (or on the local cluster) that can coordinate information retrieved from these applications and use it to determine a bandwidth estimate for all applications wishing to communicate with that remote host, the estimates from the applications are unlikely to be very accurate. For example, the applications will not be able to take into account the presence of each other unless they have some global knowledge of the overall state of communications in the local host. That is information that the operating system itself keeps.

The need for a central server for all applications to talk to also fits in nicely with the desire for the SNMP estimate to only be made once in a certain time period to each remote host. Uncoordinated applications each doing all the bandwidth estimation themselves would obviously each potentially make at least one SNMP estimate. If several of these are fired off at around the same time, the routers on the shared portion of the paths to the remote hosts will have to deal with a large SNMP processing load and the available bandwidths of the links would be reduced for other traffic.

The observed throughput from an application must be combined with the SNMP based initial estimate to give us a new estimate of the throughput to a remote host. It must be remembered that this new estimate is likely to be lower than the original estimate as that would have been based on the nominal bandwidth of the links between the routers in the path.

5.8 Implementation

A prototype SNMP-based path throughput estimation tool has been produced using the TCL-SNMP toolkit. This makes use of the interpreted Tool Command Language developed by John Ousterhout at Berkeley with special extensions to the core language to handle SNMP transactions. The interpreted nature of TCL was not a problem for this prototype as the RTDs of the links traversed in a WAN far outweigh the amount of time spent processing the information that has been retrieved. Also, only one SNMP probe was active at once[2].

[2]In a real system it would be hoped that each SNMP probe could be given a lightweight thread of its own so that all the probes can run in parallel, thus decreasing the overall delay.

An example of typical response times from the prototype from the local host `lust.mrrl.lut.ac.uk` (A Sun Sparcstation IPX running SunOS4.1.3) to the host `library.mit.edu` was found to be around one hour and ten minutes when run between midnight and 6am BST. The RTDs reported by the UNIX `ping` command were between 100 and 200ms depending on the load on the Internet at the time. The bulk of the execution time was the taken querying the router `icm-lon-1.icp.net`. This router appears to hold a large routing table containing routes to many Internet accessible networks. The initial implementation of the throughput estimation script had to retrieve over 31000 records from the router in order to make its estimates. This took well over half an hour on every trial.

To help overcome this large delay and to reduce the load that the throughput estimation would place on the network, the monitoring program was modified to cache successfully recovered nominal bandwidths between pairs of IP addresses. This reduced the program's run time on subsequent estimation attempts to the same host down to a couple of minutes. The nominal bandwidth cache lifetime is currently assumed to be infinite as it depends upon the physical path between two adjacent hops in the path across the Internet and this is assumed to be constant. It should be noted that this assumption may not be true in systems where there are multiple single hop paths between two IP hosts with different metrics, or where link speeds are upgraded. However, it must be remembered that the ICMP/SNMP based estimate is just a starting point for further estimates based on observed application-to-application throughput. If such observations suddenly started to show real throughputs that are greater than the cached nominal bandwidth of any hop in that path, the cached figures could then be recomputed if necessary[3].

It was also noted that on paths over the Internet that involved many hops[4], sometimes over half of the routers along the path failed to answer the SNMP queries. Some of the others that did start to answer queries would stop responding and cause the TCL-SNMP package that the prototype script was written in to abandon communications with those nodes. Completely non-answering routers only take 15 seconds to process (3 attempts with 5 second timeouts) but

[3]The only real reason to do this is to find a new estimate of the maximum throughput that applications could ever hope to achieve between two hosts. One could simply use the largest observed bandwidth to date as the nominal bandwidth is only a very rough upper bound on the bandwidth that an application can expect from a link.

[4]In other words, international routes, especially those across the Atlantic.

routers that start to answer and then give up can waste much more of the script's time. The reason why these nodes suddenly failed to respond is unclear; multiple attempts managed to get varying amounts of information out of them using identical queries. The most probable explanation is that either they or some routers closer to the local monitoring host are heavily loaded and so the SNMP UDP packets used for either the requests or the responses are being discarded. Another advantage of caching results is that when one of these routers does manage to produce a usable answer, it does not need to be queried again.

This prototype implementation shows that it is theoretically possible to use SNMP and ICMP to discover the nominal bandwidth available along a path through the Internet. However it also shows that SNMP querying is an expensive option, both in terms of bandwidth required, time taken, and router and host CPU utilisation. The lack of response from some routers and the apparent fragility of the UDP based protocol over congested links serve to limit its usefulness for process placement in the real world.

5.9 Multicast Placement

One possible solution to the process placement dilemma outlined above is to not attempt to determine the performance characteristics of the network at all. Instead, the system could simply send its request out to a large group of hosts and wait for the replies to be returned. This could be done using normal unicast style communications simply looping round all the known hosts, sending the request to each in turn and then having the client sit and wait for a reply to be received. However, this is not a very efficient use of the network as the for a request of n packets sent to m hosts, nm packets would be transmitted from the local machine. Also the host at the top of the client's list of target servers is likely to constantly receive the request before the host at the end, especially if the group of target machines is large. A solution to this problem is to use true multicast communications.

Multicast communications fits somewhere between the widely used unicast and broadcast communications techniques. In a unicast communication, the source host sends data to only one destination host. With broadcast, the source host sends data to all other hosts on the network or, more typically, all other hosts on the same subnet. Multicast communications

allows the source host to simultaneously send the same data to multiple hosts anywhere on the network. This set of destination hosts that a message is sent to form a *multicast group.* Hosts not in the multicast group that data is sent to do not see it or merely discard it. Multicast communications therefore allows data to be simultaneously sent to a number of hosts at a low network overhead and without unduly affecting the performance of hosts that are not concerned with the data.

Some network technologies implement multicast support at a very low level. For instance, the Switched Multi-megabit Data Service (SMDS) [35] includes the concept of multicast group address in its data-link layer. Some transport protocols, such as XTP [30, 142], also provide multicast capabilities to their clients by making use of the broadcast or multicast capabilities of the underlying network technology. XTP can use a number of different multicast addressing styles, including the Internet Class D addresses [43].

Class D addresses are also used in the rapidly growing MBONE virtual network that uses the global Internet as a physical transit network [52]. The MBONE makes use of multicast IP packets. On network technologies that support broadcast and/or multicast communications, such as Ethernet based LANs and the SMDS links, the multicast IP packets can be transmitted directly on the wire. In between these "multicast islands", the multicast IP packets are encapsulated inside normal unicast IP packets and routed through "tunnels" using the standard Internet routing mechanisms. This encapsulation is performed using multicast routers, either in software and hardware, that exist at either end of the tunnel. The topology of the virtual MBONE network is being carefully designed with the physical topology of the underlying Internet links in mind so that the same data does not traverse the same physical links more than once if possible. This careful design is an important consideration as the current usage of the MBONE is bandwidth intensive audio and video conferencing [101].

The rapid growth in the MBONE is aided by the support of a variety of systems vendors in implementing the underlying multicast IP support in their protocol stacks. To date multicast IP support is available for Sun, DEC, HP and SGI workstations and IBM PC compatibles. There is also the promise of multicast IP communications software for Macintoshes in the near future[5]. Multicast IP thus appears to be a relatively safe technology that can be used to build

[5]Indeed beta versions of Apple's OpenTransport product are apparently already available that have some

future distributed applications as it already has commercial backing and is likely to be further optimised and developed for desktop conferencing applications.

5.10 Multicast LbRPC

The multicast LbRPC mechanism proposed here is the marriage of the idea of exporting code and data to a remote host for processing with multicast communications support. It extends the basic LbRPC mechanism by allowing the code and data to be simultaneously multicast to a group of hosts as opposed to a single machine. This multicasting of code and data is in effect the process placement. No performance estimations or system status probing is required and no explicit placement decision is made.

Multicast LbRPC makes use of a client-server computing paradigm. Clients are the distributed application processes that wish to export code and data over the network for remote execution. Servers are background processes that are running on hosts distributed across the network that are prepared to take code and data from clients for evaluation. A single physical host machine can support multiple clients and servers simultaneously.

When an application wishes to export code and data for remote evaluation it makes a call to an exportation routine on the local host. This call is implemented using either a language extension or library routine that takes both the code and data, and prepares it for transport over the network. This preparation may be as simple as marshaling the code and data into buffers for transmission or it may involve more complex format conversion and the options that were covered in Chapters 3 and 4.

The exportation routine is also supplied with a Class D address and a port number that is used to transmit code and data. This <address, port> pair may either be a "well known" service or it may be determined dynamically using a session directory mechanism, such as sd [80]. It uses the <address, port> pair to open a communications stream over the Internet and then multicasts the marshaled code and data. The client now has a choice; either it can block waiting for the results of the LbRPC call to be returned or it can continue local processing asynchronously. Asynchronous processing appears to offer the opportunity to achieve some

---

limited multicast capability.

coarse parallelism and may be useful in some applications for "soaking up" the inevitable delay caused by link latency.

However, asynchronous processing is more complicated to implement than the blocking style of communications as the system must keep track of the state of the local process that may be affected by the results of the execution of the code on the remote host. An obvious example is a variable used on the right hand side of an expression that may also be altered in the exported code. There are also more subtle possibilities in some programming languages, especially those like C that allow multiple pointers to point to the same data structures in memory. The Amoeba distributed system found that asynchronous processing of its RPC mechanism did not justify the complexity of its implementation [173] and so the possibility of asynchronous LbRPC transactions should be an optional implementation detail.

5.11 TCL-DP Prototype Implementation

The prototype implementation of multicast LbRPC has been built on top of the TCL-DP package. TCL-DP [159] is a distributed programming extension to John Ousterhout's Tool Command Language (TCL) [124] developed by Brian C. Smith and Lawrence A. Rowe. TCL-DP was an obvious choice for developing a prototype multicast LbRPC implementation as it already provides the ability to export code and data for evaluation on a remote host. The unicast TCL-DP can achieve this exportation as TCL is an interpreted language heavily slanted towards string handling and so can easily pass copies of TCL source code to the remote host that can be evaluated in a remote interpreter and have the interpreter's result string returned to the calling client. However, this ease of implementation does not imply that TCL is an ideal language for fulfilling LbRPC's more general goal of providing an intermediate exportable code format as was discussed in Chapter 3.

5.11.1 Basic Multicast Hooks

The first addition to the TCL-DP package was to add in low level support for multicast network communications[6]. This was achieved by adding a new option to the `dp_connect` command. The

[6]The prototype code described here was made available to the Internet community and has now been adapted by Rowe and Smith to form an integral part of more recent TCL-DP releases.

new option, -mudp, requests the construction of a multicast UDP socket to be created. A Class D Internet address has to be specified, along with a port number and a multicast Time To Live (TTL). TCL-DP code that makes use of this new option causes the invocation of a new C function in the TCL-DP interpreter ("dpwish" or "dptcl") that creates the socket, checks that the address supplied is a Class D address and then sets the required multicast IP socket options before binding the socket to the multicast address.

The socket options specified include setting the multicast TTL (thus allowing the user application to restrict the spread of a request to a multicast group to a subset of all the hosts in the group based on their topological proximity in the MBONE), socket reuse (to allow multiple multicast LbRPC clients and servers to bind to the same port) and multicast IP loopback. The last option is interesting as it means that a server running on the local host will receive any requests transmitted by the local clients. This means that a request from a local client can be guaranteed to always return some form of result by ensuring that at least a local server is listening to the same multicast group as the client is transmitting requests on. It also means that near optimum response time is maintained if the code is best suited to being executed locally. This is due to either architectural dependencies, heavy loading of remote hosts or intervening network performance.

One point that came to light during the development of this part of the code was that different versions of the UNIX operating systems handle binding to multicast sockets differently. This point is not heavily documented for multicast IP programmers and may well cause problems for some multicast code that is developed on one style of platform and then ported to another.

5.11.2 Multicast LbRPC Mechanism

Once the basic dp_connect command was capable of multicast communications, a simple Multicast LbRPC (MLbRPC) mechanism was built on top of it using a TCL-DP script for both the server and the client. The mechanism marks every MLbRPC request with a Transaction ID (TID). The TID is a globally unique string devised by the client that identifies a particular request. Generation of the TID is implementation dependent and relies on information such as the current time, the process ID of the process containing the calling TCL interpreter, a

sequence number and some host specific ID.

The TID allows the client to re-transmit the same request without fear of the same remote hosts bothering to execute it. This facility may be useful for occasions when no replies have been received due to network failures even though one or more remote hosts have actually processed the data. A server that receives a request with a TID that it already has a cached result for simply returns the precomputed answer. The cached results can be destroyed by the remote host after a timeout specified by the local host.

In the prototype implementation the client also sends its unicast IP address and a dynamically generated port number. This data allows the server to reply directly to the requesting client without disturbing the other hosts in the multicast group. Results could be returned via the multicast group, allowing servers to see when a result has already been produced and therefore abandon their own work as it may now not be required. The problem with this feedback style is that the result may not make it to the original source host due to a network failure. Thus the retransmission of the results by another host might save another high latency request for the same code from being sent by the source host. If it were used it would also mean that the servers would have to distinguish between requests and responses.

However, the calling client must also maintain a record of the IP addresses of hosts that have already replied. This data is required to prevent multiple responses being accepted from the same server for the same transaction due to retries when more than one response is desired. There is little point in recording the response from the same host more than once; it must be assumed that if the application programmers indicated that they would like to receive more than one reply then they would like the replies to come from different servers.

There is also the danger that a "melt-down" point may be reached with unicast replies from multiple servers hitting the source host and preventing it, or other machines that it shares its LAN with, from transmitting or receiving other packets. Multicasting a "all the results needed have now been received" packet from the source host to all the servers in the same multicast group after the required number of answers had been received would limit the effect of the pathological case where only one result is required but hundreds of machines try to run the code. Multicasting the results back to the calling host would also work as the other servers could also see it, but the request would then need to have the number of desired replies encoded

in it, and, as was mentioned above, there is no guarantee that the original source host saw the reply even if all the remote hosts did.

In the first prototype implementation a side effect of the support for only unicast replies is that it is possible for the MLbRPC programmer to process more than one response for a single transaction on purpose. This side effect makes MLbRPC a more focused and powerful version of some of the traditional broadcast RPC mechanisms that some vendors support [157, pages 5-19–5-20]. However it does tend to call into question the whole remote *procedure* call programming paradigm [171].

5.11.3 Performance

To test the performance of the first prototype implementation of MLbRPC, a number of fragments of TCL code were executed in a variety of configurations. These code fragments were:

- A null operation:

  ```
  {}
  ```

- A simple variable instantiation:

  ```
  set colour "green"
  ```

- A simple regular expression substitution on the string "This is a trial of the new Multicast Late-Binding RPC" held in the variable `text`:

  ```
  regsub "new" $text "great new" newtext
  ```

- A more complex regular expression substitution, including reading in a file on the remote machine (the TCL-DP 3.2 announcement text file):

  ```
  set inf [open "ANNOUNCE" r]
  set text [read $inf]
  close $inf
  regsub -all -nocase {Tcl[- ]DP} $text "Multicast Tcl-DP" newtext
  ```

- A simple mathematics function for finding a number raised to a power. This uses an exported procedure definition and, due to the large power being raised to in the test, several thousand iterations.

Operation	*Locally Executed on 386*	*Exported to Indy*
Null Op.	69μs	43016μs
Simple `set`	970μs	46369μs
Simple `regsub`	7512μs	61599μs
Complex `regsub`	1513698 μs	81836μs
Simple maths function	107042583μs	3695837μs

Table 5.1: Performance comparison of the non-exported TCL against MLbRPC executed code using interpreted TCL code on the client side

```
proc power {base p} {
set result 1
while {$p>0} {
set result [expr $result*$base]
set p [expr $p-1]
}
return $result
}
power 1.01 5000
```

Each of these code fragments was executed five hundred times and the average execution time determined using the operating system's time and process resource usage system calls on the calling client. The results for the first prototype MLbRPC implementation are shown in Table 5.1. The locally executed code was running on an ISA based 386DX33 PC running Linux 1.3.97 and the remote code was exported to Silicon Graphics Indies with 32Mb of memory running IRIX 5.2, connected by an otherwise unloaded Ethernet. The machines were only running standard background tasks at the time of the tests; there were no other user processes running.

The first prototype version of the MLbRPC code uses TCL based procedures on both the client and the remote server to handle the exportation of the code and data. The performance of this system is poor for small fragments of code. Indeed as can be seen from Table 5.1 that the

Operation	*Locally Executed on 386*	*Exported to Indy*
Null Op.	69μs	19056μs
Simple `set`	970μs	20417μs
Simple `regsub`	7512μs	22504μs
Complex `regsub`	1513698 μs	54422μs
Simple maths function	107042583μs	3652763μs

Table 5.2: Performance comparison of the non-exported TCL against MLbRPC executed code running over LANs with the MLbRPC mechanism implemented using built in C functions

processing costs of a standalone TCL version of the first four exported sections of code are about four or five orders of magnitude faster than the same code exported to remote hosts, regardless of whether they are on the same LAN segment or another segment on the campus network. However, the fifth code fragment, a simple mathematical function that iterates through a tight loop five thousand times, shows that the cost of network communications relative to the absolute time taken falls as the total remote processing time for the code increases. This saving in relative costs for longer running code is in line with the concepts of late-binding based on theoretical predictions.

This points out the fact that even with multicast placement, there is still a minimum processing time requirement to make remote placement worthwhile. Unfortunately this figure can not easily be computed as it is again dependent upon the variables that would be required by the process placement algorithms that MLbRPC is intended to replace. However, by not computing a complex and unreliable placement algorithm, a potentially significant processing load on the source host has been removed. If the local machine also attempted to compute the result in parallel with the code exportation it would ensure near optimal results.

As these results were somewhat disappointing for smaller code sizes, the client side routines were converted from TCL procedures into built in commands in the `dptcl` shell. This conversion was done in two stages; firstly the routine that generated the unique TID and then the full client execution command. The results for this new version running the same machines over the unloaded LAN are shown in Table 5.2.

Operation	*Locally Executed on 386*	*Exported over Internet to Alpha*
Null Op.	69μs	963426μs
Simple `set`	970μs	3565316μs
Simple `regsub`	7512μs	1357977μs
Complex `regsub`	1513698 μs	1297416μs
Simple maths function	107042583μs	8202940μs

Table 5.3: Results of MLbRPC code and data exportation to a DEC Alpha over the Internet, compared to running the code locally on a 386 PC

The results show that the performance improves greatly by using compiled rather than interpreted code. Indeed, the complex `regexp` routine, that in itself is not terribly complex when compared to many "real world" TCL scripts, now only runs roughly half as fast even on the local LAN with machines of similar performance acting as both client and server. In that respect these trials were worst case tests as there was no way the remote host could perform better than the local host unless the local host was very heavily loaded. These results bode well for future implementations that would make use of specialist computational engines, such as supercomputers, vector processors or database servers, where the potential speed up of remote processing over local processing can be much greater. It must also be remembered that TCL is not an ideal intermediate code representation [94]; it is used here purely because it made prototyping faster and easier.

Lastly, the MLbRPC prototype system was ported to OSF/1 and run on a DEC Alpha 4000/710 in Palo Alto [7]. Table 5.3 shows the results of this trial. The experiment was performed at around noon in the UK, 4am in Palo Alto, with the Alpha's average number of jobs in the run queue hovering around six, including some very long running CPU bound jobs. The Alpha was ten hops over the Internet from the the source machine, and the local LANs were in normal production use. This environment can therefore be considered to be a fairly realistic example of a real WADCS, although the network links are not running at the gigabit per second speeds.

[7]This machine is one of the Alpha systems made available to the Internet community by Digital for use in porting and product testing. See <URL:http://www.digital.com/info/alpha-demo.html> for more information.

From these results it can be seen that even when running over the production Internet, with all its usual packet loss and congestion, a sufficiently fast machine such as the high end Alpha can still out perform underpowered local workstations when running complex, CPU intensive operations. However note that even though the Alpha based machine is much faster than the Indies that were used for the local exportation experiments, a combination of heavy loading and packet loss means that a local Indy would still execute the last two code fragments fastest. If the Indies had been listening for MLbRPC requests at the same time that the Alpha tests were conducted, they would have answered first and the application would have got the optimal response. This demonstrates that MLbRPC does offer potential optimal speed ups for some classes of code (the long running, CPU intensive cases).

5.12 Discussion

One problem that is immediately apparent when considering MLbRPC for applications such as distributed database queries and updates is how a user goes about handling non-idempotent transactions in a multicast environment. A non-idempotent transaction is one where multiple executions of the code gives different results from a single instantiation. If it is only the repeated execution of the same code and data on a *single* host that is of concern there is little difficulty in ensuring idempotency as the TID can be used to see if the request has already been satisfied and the cached answer is returned instead. However, if the exported code and data should only be executed once anywhere on the network then there is a larger problem.

In this case it must be possible for the processing hosts to run the exported code in an isolated environment and only commit any changes that it makes into the shared, global environment when they are sure that no other hosts on the network can do the same. The "environment" in this case includes not only the data exported along with the code from the local source host, but also any input and output on the remote host that the executing program produced. As multiple hosts are likely to be processing the code simultaneously, only the first remote host to reply to the source host should commit its environmental changes. All other remote hosts should be able to roll back their processing so that the environment is in the same state as it was before they began execution.

To solve this problem the client should be able to indicate the "chosen" remote host that has to commit the operation. This host is the first one to return the results of the computation to the client. The indication can take the form of another multicast transmission from the client indicating that the remote host is to commit the results for the specified TID. Other remote hosts listening to the same multicast group would then discard their results without affecting their environment. The chosen remote host should send a response to the client when it has committed the results. This response can be unicast if none of the other hosts need to know when the commit operation has been completed. Provision would need to be made for retransmission of the commit requests and responses to allow the protocol to work in network environments with high packet losses.

One side effect of having this commit protocol built into the MLbRPC mechanism is that a full MLbRPC transaction now requires at least two full round trips rather than one; longer if there is high packet loss. The client side could continue processing with the returned results asynchronously with the second phase of the commit protocol but then this introduces failure modes where the communications to or from the remote host is lost before the local hosts knows for certain that the commit operation has succeeded.

As latency is the fundamental constraint that LbRPC is trying to overcome by maximising the amount of work done in each round trip, a two phase commit protocol is obviously not a good thing. However, other alternatives such as disallowing operations in the exported code that are likely to create idempotency problems or leaving commit protocol design to the application programmer are also not desirable as they encroach upon the network transparency that the RPC paradigm is supposed to be supporting.

If the implementation allows MLbRPC calls to commit successfully on multiple hosts one must also look at how this relates to the basic RPC abstraction. This facility is much the same as the extension that some vendors have made to deployed traditional RPC mechanisms to allow the use of broadcast addresses. In both cases the call no longer invokes a single procedure as it would do in a non-networked environment. The programmer could view the MLbRPC request that allows multiple responses to be returned to be a form of "meta-procedure call" that is calling a non-existent metaprocedure that calls all the actual procedures with the same arguments, waits for the given number of responses or the timeout period and then marshals

all the separate responses into a single response for the calling routine. This view allows the applications programmer to still think in serialised, procedural terms that is one of the main goals of the whole RPC paradigm.

Unfortunately, a MLbRPC request that can have multiple responses would have fairly complex failure semantics. The system would potentially need to differentiate between a failure to supply the requested number of responses due to lack of remote hosts in the chosen multicast group and an actual failure of the remote processing due to bugs in the exported code. The prototype overcomes the first of these by implying that the requested number of responses is the *maximum* number of replies that the application programmer wishes to receive. It is up to the programmer to deal with situations where the application gets fewer responses than desired. Padding the returned list of results with null responses to fill up to the requested number of responses is an alternative, but this would then require a mechanism to differentiate between automatically inserted padding nulls and real null responses from remote computations.

In the prototype implementation, it has been implicitly assumed that if the client can find MLbRPC servers that will take the exported code and data and return the results, it has the permission to perform the computation on those remote hosts. The ability of MLbRPC (or in fact any LbRPC style mechanism) to allow remote hosts, possibly widely distributed over an Internet, to run arbitrary code on a host raises a number of security issues. Discussion of these will be deferred until Chapter 7.

One last point is that the prototype's use of raw UDP packets for transporting the requests and results is not good enough for a production system. Firstly both the request and the response must fit into a single UDP packet, placeing limits on the procedures that can be exported in this system. Secondly, UDP is inherently unreliable and packets are often dropped by routers in the Internet at times of congestion ahead of packets from other transport services. MLbRPC should really use a multicast reliable, multi-packet transport protocol for the requests and possibly a lightweight reliable streaming protocol for the results. The next chapter will look at the providing transport services to support MLbRPC systems in more depth.

5.13 Conclusions

This chapter has outlined the idea of combining a multicast network protocol with the Late-binding RPC paradigm suggested by Craig Partridge as a means of placing exported code and data in a WADCS.

Bandwidth estimation was shown to be a non-trivial problem that needs to be approached if forthcoming distributed applications and networked information retrieval tools are to make intelligent decisions on their use of the network. This chapter has detailed the way that the existing Internet infrastructure and protocols such as ICMP and SNMP can be combined with application observed throughputs to allow estimates of the throughput that the applications can expect to achieve.

The upper bound on the throughout an application is able to achieve is easy to determine as it is related to the nominal bandwidth on the local network link. However communication paths to remote hosts over the Internet are likely to have far lower effective throughputs as the traffic from the application on the local host will have to contend with traffic from a large number of other, independent sources.

A simple method to make throughput estimations to remote hosts has been proposed based on the nominal bandwidths of links discovered using a combination of ICMP and SNMP probes and heuristics from actual communications sessions between the local and remote hosts. It has been argued that if this throughput mechanism is used, the job of performing the ICMP and SNMP probes and the actual estimation should be performed by a separate application level server on either the local machine or machine in the local cluster. This server can then handle multiple requests for bandwidth estimates to the same remote host from a number local applications without increasing the load on the whole Internet over that required by a single application.

However, the use of ICMP and SNMP is quite slow and resource intensive. If used at all it should only be done once for each remote host. If there is a relatively small working set of hosts that the local applications can talk to, the overall increase in traffic from that host or cluster of hosts need not be very great. Nominal bandwidth figures are cached, reducing response time on subsequent probes along paths that include hops that have already been investigated. Unfortunately it was also shown that it is often impossible to retrieve any useful data from

remote routers and hosts if the administrative policy of those that run disallows remote SNMP queries.

Multicast Late-binding RPC appears to offer an acceptable alternative to trying to second guess the performance of the hosts and networks in trying to make a process placement decision. Although the early prototype is fairly slow, the mechanism does guarantee that, if distributed services are used at all, the fastest elapsed time possible is achieved. It does this at the expense of some wasted bandwidth and CPU time, but these are expected to be plentiful in future network environments and certainly not as much of a problem as the fixed minimum latencies suffered by high speed WANs and internets.

This chapter has detailed the design and implementation of a prototype MLbRPC system based upon TCL-DP. It has shown that MLbRPC is indeed possible and has the potential to offer the near optimal performance from exported code when it is executed in parallel on both the local source host and a number of remote machines. It does however introduce some problems, mainly concerned with the thorny issue of idempotent request processing that often plagues distributed systems. The use of raw UDP packets as a transport mechanism in the prototype also revealed the need for a lightweight multicast transport mechanism that permits sequenced streams of code and data to be sent to the remote machines. This last element will be investigated further in the next chapter.

Chapter 6

Transport Protocols

6.1 Introduction

Multicast communication mechanisms are becoming more widely used in the Internet with the introduction of Multicast IP [43], the MBONE [101, 44] and new WAN technologies such as SMDS [35] that support multicast communications natively. There is reason to believe that these mechanisms may form a useful way of performing code and data placement in widely dispersed distributed systems built on top of the global Internet infrastructure as was shown in the previous chapter. However, to do so there must be a Multicast Transport Protocol (MTP) available that is able to provide the necessary support for the request-response style of interaction that is common in distributed systems whilst minimizing the number of round-trip delays incurred by each transaction.

This chapter describes the development of a simple multicast transport protocol designed to support the needs of MLbRPC described in the last chapter It is intented to work over large internetworks.

The proof of concept prototype of the MLbRPC that was built and detailed in the Chapter 5 utilised raw multicast UDP packets as its transport layer, limiting its usefulness due to the need to constrain the size of the code and data being exported to fit in a single UDP packet. If the maximum amount of work possible is to be achieved per network transit [131], it is clearly desirable to be able to send much larger streams of code, data and results to and from the remote MLbRPC servers in a WAN. The transport protocol proposed later in this chapter

achieves this goal and is optimised to support the particular communications needs of the MLbRPC mechanism.

In this document there is an initial overview of some of the existing MTPs, which outlines some of the lessons that may be learned from them. The needs of MLbRPC as far as MTPs are concerned are then outlined, followed by the design of a new MTP specifically intended to support it. Lastly some results from a prototype implementation are presented and some conclusions are drawn.

6.2 Previous Multicast Transport Protocols

The transport layer is the fourth layer of the ISO Open Systems Interconnection (OSI) seven layer reference model. The transport layer is designed to add functionality such as reliability, sequencing and connection management to the lower network layer support. There have been quite a few MTPs proposed and produced in the last ten years.

6.2.1 Unreliable vs. Reliable

MTPs can be partitioned into different groups in many ways. One categorisation is to split them into two broad types; *unreliable* and *reliable*. An example of the first these is the use of the User Datagram Protocol (UDP) [139] when used over Class D IP addresses. UDP offers very little over the raw network service and makes no guarantees as to the delivery of the data at the remote site. As its name suggests, it is also based on connectionless datagrams rather than a connection oriented stream. Thus higher application layers are responsible for handling the segmentation of data streams larger than one UDP packet and providing any required error handlers. Even so, many multicast applications in general usage on the MBONE are based on UDP. They work because they utilize human users' ability to extract useful information from multimedia data streams that experience a high rate of packet loss in passing through the network. Examples of applications in this category are the network video tool *nv* [57] and the visual audio tool *vat* [81].

The Versatile Message Transaction Protocol (VMTP) [27, 28] which provides a reliable unicast transport protocol to support Remote Procedure Calls (RPCs) used in the V distributed

system [29] also supports unreliable multicast transportation. This unreliable multicast service is handled by making direct use of the underlying Ethernet [112] hardware upon which VMTP was designed to be used. VMTP provides support for packet retransmissions, duplicate packet detection, and security in its unicast mode, but leaves reliable multicast communications to higher layer protocols. The rationale behind this is that the applications that require multicast support from VMTP can build reliability in using *positive acknowledgments.*

6.2.2 Positive Acknowledgments

Positive acknowledgments are commonly used to provide reliable communications in unicast transport protocols [140, 54, 118] but can cause scaling problems in MTPs. This is because a positive acknowledgment based protocol[1] is required to send an acknowledgment indication for every data packet that is successfully received. To make these types of protocols slightly more efficient, most positive acknowledgment protocols allow blocks formed from a number of packets to be acknowledged in one go. This increases the amount of buffering required in the sender and requires the negotiation of a sensible *window* size.

Scaling problems in positively acknowledged multicast protocols occur for a number of reasons. Firstly, as the number of receivers increases, the chance of an *implosion* occurring also increases. An implosion occurs in a multicast protocol when a large number of replies from remote machines are received in response to a single multicast request in a short time frame. This limits the scalability of the protocol as the local network to which the client is connected becomes saturated with reply packets, the client itself may collapse under the weight of packet processing and collision avoidance mechanisms of some network protocols can cause a positive feedback loop to occur resulting in a "network meltdown".

Positive acknowledgment protocols also require state to be kept for each receiver that the sender knows about. In some other positively acknowledged protocols, including the ISIS programming environment's multicast protocols [11] and Jon Crowcroft's TCP-like MTP [39], this means that any host wishing to become a sender in a multicast group must somehow acquire a list of all the group members and their state information before it can multicast any data. ISIS defined an explicit mechanism for determining group membership but unfortunately

[1] Also known as a "Stop and Wait" protocol.

this required large amounts of state information to be transferred which reduced its scalability (although its use of TCP streams to provide a form of reliable network protocol to support the multicast also contributed heavily to its poor scaling characteristics).

6.2.3 Negative Acknowledgments

One alternative to positive acknowledgments is to do the exact opposite; instead of sending an acknowledgment when a packet is successfully received, the receiver only sends acknowledgments for missing packets. This is results in *negative acknowledgment protocols.* Negative acknowledgment protocols can either use a timer to determine when a packet should have been seen or check to see whether there is a gap in the sequence numbers. The negative acknowledgment indications can be generated either as soon as a missing packet is detected or after waiting a short while to allow for misordered packets to be received. For a fairly reliable network service, a negative acknowledgment protocol is less likely to cause an implosion effect than a positive acknowledgment protocol as the number of missing packets requiring a negative acknowledgment indication would be outweighed by the number of successfully received protocols the *did not* need to be acknowledged.

There are some problems with negatively acknowledged reliable MTPs. Firstly, negative acknowledgments do not lend themselves to providing flow control and can easily allow receivers to be overrun by fast senders. The usual means of working around this limitation is to send information on the state of a receiver's buffer space or rate limiting requests when it does send a negative acknowledgment so that the sender adjusts its rate when it realises things are going wrong. Of course, if the underlying network is unreliable there is no guarantee that the negative acknowledgment will get through and the sender will still charge ahead, leading to a positive feedback loop (the lack of a negative acknowledgment packet makes the sender think that the receiver is handling all the packets it is transmitting which in turn results in it sending more, causing further negative acknowledgment indications to be lost due to packet loss at the receiver or intermediate congested nodes in the network).

6.2.4 Saturation Protocols

Another means of achieving reliability in MTP design is to cut out all acknowledgments. Instead, every packet is transmitted more than once and the probability of at least one copy of every packet getting through is relied upon. This is termed *saturation* and works by virtue of the fact that the network is unlikely to drop the same packets every time. Obviously, the more times the packets are retransmitted, the higher the chance that the receiver will acquire copies of all of them are. Obviously saturation protocols can waste large numbers of packets but they do not need to stop and wait like positive acknowledgment protocols. They are especially suitable in situations where the number of receivers is unknown and potentially very large and dynamic and the size of the data being sent is known at the start of the transmission. They also benefit from the request-response transaction communication style, where the responses indicate that at least some remote hosts have successfully received the request and processed the transmitted data. The large group multicast transport protocol by Jones et al [87] uses saturation transmission for small messages, whilst switching to negative acknowledgments for larger messages to keep the number of redundant, retransmitted packets down.

6.2.5 Fault Tolerance

Some multicast protocols[188, 115] make use of a token or a central site to mediate which host in a group is able to initiate a multicast communication. This can help prevent implosions and ensures that slow hosts are not left behind by fast ones. Unfortunately this introduces a single point of failure into the system which is not desirable, especially when the communications are taking part over unreliable network links. The loss of the token or central site can either stop the protocol completely or cause it to slow down whilst the other members of the group detect the error and regenerate the token or re-elect a new central site.

An example of a token passing protocol is the Reliable Multicast Protocol (RMP) [188, 115]. RMP works well in small groups when used over LANs but problems have been found in using it over the Internet. In the latter case membership of the multicast RMP group can become disjoint and attempting to repair the virtual token ring can result in mutually exclusive groups each with separate tokens sharing a single multicast IP group. Work on solving these problems with RMP is still ongoing.

6.2.6 Lessons Learnt

A number of lessons can be learnt from the above review of existing MTPs:

- Positive acknowledgments are likely to cause scaling problems in MTPs due to the potential for implosions at the originating host caused by many responses being received at the same time and the need for the sender to keep state information for each of the receivers.
- Negative acknowledgment protocols can suffer from the lack of inherent flow control causing the sender to run ahead of the receiver.
- Saturation protocols are the least likely to cause an implosion at the sender as there are no acknowledgment packets returned from the receivers, although it is still possible for a set of simultaneous responses to be generated. If the communication is occurring over WAN links to a variety of different hosts, the likelihood of simultaneous responses being received is also greatly reduced.
- MTPs that rely upon the presence of a single site or a token passed between the group members are more likely to fail than protocols that are fully distributed. Although this is true of all networks, it is far more likely to affect systems built over WAN internets than LAN based systems as the WAN internets have far more elements that may fail and partition the central site from at least some of it's clients.
- The design criteria applied to protocols designed to support human scale communication patterns and restrictions are likely to be different to those needed for computer-to-computer communications. Specifically, human users are capable of enduring quite high levels of packet loss in multimedia data services whilst still being able to extract useful information from the system.

6.3 Designing an MTP to support MLbRPC

This section investigates the possibility of an MTP specifically designed to support the use MLbRPC systems over WANs. The needs of a sequenced, semi-reliable transport protocol for use in systems such as MLbRPC is somewhat different to the requirements that drove the

development of many of the transport protocols described earlier. One of the general precepts of LbRPC is that it should minimise the number of round-trip delays that an application has to make. Other assumptions made about the environment that LbRPC is used in state that bandwidth is cheap, computational power is plentiful and the error rate of the underlying network is low. The last point is quite important as it implies that transport protocols can be "success oriented" [96].

6.3.1 Acknowledgments, Windows and Saturation

The use of acknowledgments in existing protocols usually entails some performance drop due to the need for the sender to wait for acknowledgments from the receiver for packets already transmitted before continuing to transmit new packets. In TCP for example, the window size is limited to a 16 bit value, which is a 64K packet (or 64 1K packets). Over networks with high bandwidth-delay products (sometimes called "long fat networks" or LFNs[78]) this can severely limit throughput. The protocol is forced to suffer multiple round trip delay time pauses whilst it waits for acknowledgment packets. This goes against the grain of the "minimise the number of round-trip delays" condition of LbRPC.

Acknowledgments and windows are used in transport protocols for two main reasons; to detect packets dropped by an unreliable network protocol and to prevent the receiver's buffer from overflowing. The former is a very real problem in existing wide area internetworks as the Bit Error Rate (BER) can be quite high on some links and congestion can cause intermediate nodes, such as routers, to drop packets due to resource starvation. The high performance networks of the future are likely to be less susceptible to this problem as they will have lower natural BERs due to the use of fibre optic technology and will implement congestion avoidance techniques in cell based lower layers (such as the simple Leaky Bucket and the Token Bucket algorithms described in [132]).

The problem of receiver overflow is one that is becoming less important with time as memory sizes and processor speeds improve. Even protocols using large sliding windows will require large buffer spaces[2]. As the MLbRPC mechanism is being used to export code and data from a local program to a remote machine for execution it is realistic to assume that a buffer several

[2]TCP with large windows permits the window to grow to several megabytes for example.

megabytes in size will be more than sufficient for most tasks. The receiver can allocate a large enough buffer for the exported request if *all* the packets in the request carry an indication of the total size of the request. Again, this is possible in this particular application as the request size is known by the MLbRPC system before any network traffic is generated, something that is not normally possible in general purpose streaming transport protocols.

Therefore the protocol used to support MLbRPC may benefit from the removal of all acknowledgment and sliding window elements found in other transport protocols. It must also be remembered that the multicast "channel" is only required for the outward leg of the communication, exporting the code and data from the local client machine to the remote MLbRPC servers. The results from the remote MLbRPC servers can be returned to the client using a unicast stream from each server to the client as was explained in the last chapter. This differs from many existing multicast transport protocols that attempt to provide bidirectional multi party reliable multicast communications.

Another difference in the specific application domain of MLbRPC over a more general environment is that there are potentially a very large number of servers present in the network that will receive the request but only a relatively small number will actually reply. This occurs for a number of reasons. Firstly, it is unlikely that the owners of machines running MLbRPC servers will allow just anyone to connect to their servers anonymously and run computational jobs. Instead it is likely that user will be required to be authenticated in some way before their jobs will be processed. Each user is likely to only have authorisation to make use of a small number of remote hosts and servers for which a user does not have the required permissions can quietly ignore any packets in any transactions from that user.

On the other hand, some users may not wish to have their code executed or even readable by any random servers out in the network. To prevent this they might encrypt the exported code and data using an encryption mechanism whose key is only known by "friendly" MLbRPC servers. Other servers will not be able to decrypt the request and so will be unable to generate a reply. This will be investigated in more depth in the next chapter.

Lastly, wide area networks being what they are, some links will always be down, cutting some servers off from the client and preventing them from seeing all or part of a transaction request. These servers will obviously be unable to send a reply to the client and will have to

eventually throw any partially received requests away.

Therefore the local client is likely to only receive responses from a relatively small subset of the available MLbRPC servers listening to a particular <group,port> pair. The remote servers are also likely to produce wide variations in the times that replies are received due the geographical separation and range of performance that can be expected to be available. All of these conditions combine to lessen the chance of implosions occurring, and make the saturation style protocol more attractive for this application.

6.3.2 Return Path

Once saturation has been selected as the means of providing the multicast transmission, the returned results must be considered. It would be possible to multicast these back to the same group as the request came in on, but this is not a good idea for a number of reasons. Firstly, servers and clients would have to be able to distinguish between requests and responses. This means that the MTP packet header would need to carry some means of distinguishing between the two. More importantly, every host would have to look at every packet. If multicast is only used for the outgoing request side of the transaction, the clients need not bother to process the multicast traffic themselves. Having the responses sent round via multicast is also very wasteful of bandwidth as only the client that originally generated the request is really interested in the response. The only obvious reason for using multicast to send the replies back is so that other remote servers that are still processing requests can tell when a request has already been answered. However this is not terribly useful as they have no knowledge of the number of responses the originating client required (unless this is included in the MTP packet header) or whether the multicast transmissions that they received were also successfully received by the originating client.

Therefore the return path for the results should use a unicast communications mechanism. One could use an existing unicast transport protocol for this, but many of these will incur relatively large connection setup delays. As the multicast outgoing side of the MTP is really an extended version of a normal unicast transport protocol, it should be possible to reuse some of its mechanisms on the unicast return path.

Once again, as the results will have to be generated by a remote server before any packets

are placed on the network, it is possible to include an indication of the total size of the response in all the packets returned to the originating client by each server. This allows the client to allocate the required amount of buffer space for each response.

6.4 MADTP Implementation

This section overviews some of the implementation details of an MTP specifically designed to support MLbRPC style systems. The prototype has been dubbed the *Multicast Asynchronous Datagram Transport Protocol* (MADTP) due the fact that the prototype is built upon the UDP multicast datagram service and uses no acknowledgments or sliding windows to synchronize the originating client and the multiple receiving servers.

6.4.1 Sequence number space

One problem with designing a protocol in which there is no sliding window of sequence numbers is that storing either the sequence numbers seen or those that are missing can potentially use up a lot of storage. In previous protocols this has not been a problem as the sliding window has prevented the number of sequence numbers that are in the "working set" from growing too large.

To overcome this problem it is proposed that the implementation maintain a doubly linked list of the ranges of sequence numbers that are missing. When a packet which is part of a new transmission if found, this list is created as part of the transmission's support data structures. If the new packet received is indeed the first packet of the transmission, then the list contains a single entry which is the range from the second packet to the last. If the packet that is received has any other sequence number, the initial link list contains two elements; one specifying the range of missing packets from the initial packet to one before the received packet and the other giving the range from the sequence number after the received packet to the end of the sequence space. The last sequence number in the transmission is the length of the transmission in packets and is specified in every MADTP packet header.

Using this structure, the worst case scenario occurs when every other packet is missing. Thus for a transaction of length n packets, the maximum number of linked list structures used would

be $\frac{n}{2}$. However, it is highly unlikely that the worst case scenario will be found in reasonably large transactions (which is what concerns us) as errors in the network tend to knock out whole streams of adjacent packets rather than every other packet. Small streams will consume little memory even if the worst case scenario occurs.

6.4.2 Transaction Identifiers

The MLbRPC prototype contains the concept of transaction identifiers (also known as TIDs) which are strings used to uniquely represent a particular request/response pair originating from a particular machine. These are used by the MLbRPC system to allow the servers to cache the results of computations so that if a response does not reach the client and the request is retransmitted, the server does not need to recompute the answer. In the MADTP, the TIDs are also used to allow the host receiving a packet to determine which transaction the datagram is a part of.

As such the TID has to be included in the header of every packet sent using MADTP. The TIDs used in the earlier UDP based MLbRPC prototype consisted of a 28 octet string containing the UNIX hostid [168], the time the TID was generated, the process ID of the process generating the TID and a monotonically increasing TID request counter. 28 octets is a rather large value to include in the header of every packet and so MADTP includes only a 4 byte space for the TID. Uniqueness is still preserved as the header also has to contain the unicast IP address and port to which the server must reply. The extra 4 octets allocated specifically for the TID contain a 2 octet process ID and a 2 octet monotonically increasing counter. The IP address in the header limits the transaction to a single machine and the port, process ID and counter serve to provide a value with a sufficiently large roll over value to uniquely identify the transaction itself.

6.4.3 Header Format

The header used by the prototype implementation of MADTP is the same for both the multicast and unicast sides of the protocol. The format is shown in Figure 6-1.

This gives a header which is 26 octets long. The prototype implementation carries 1K segments of the request in every packet and so the packet header increases the packet length

Offset	LSB	MSB
0	Version	Flags
2	Unicast IP	
4	Address	
6	Unicast Port Number	
8	Transaction	
10	Identifier	
12	User ID of Originator	
14	Packet	
16	Sequence Number	
18	Timeout between	
20	packet sequences	
22	Total Number of Packets	
24	in Sequence	

Figure 6-1: MADTP Packet Header Format

by roughly 2.54%. The 1K payload size was chosen so as to minimise the processing overhead whilst minimising the probability of packet loss. A smaller packet would be less likely to contain an error for a given link Bit Error Rate (BER) but a larger number would need to be sent to convey the same information and the overhead of the packet headers would be higher. The worst case size of the missing packet data structures in the receivers would also be correspondingly larger for smaller packet sizes. This particular packet size also happens to conveniently fit inside a single Ethernet packet when the IP and UDP headers are attached, which is useful if any of the hosts taking part in the MLbRPC transaction are not yet connected to fast ATM or FDDI networks. This is likely to be the case for some years to come, especially considering recent developments in switched Ethernet and 100BaseT Ethernet for the final link to workstations. In conjunction with the 32 bit sequence number in the MADTP packet header, a 1K data space size gives a maximum code and data size of 2^{42} bits.

The version octet allows multiple versions of MADTP to be deployed at once. The prototype implementation has a version number of zero; future revisions to the protocol will increase this field. Servers which receive MADTP packets which contain version numbers of the protocol other than those that they understand will simply junk the packet without bothering to attempt

to record any information. The flag octet is provided to allow originating client to signal any special information to the remote servers. In the current prototype it is currently unused and set to zero.

6.4.4 Receiving a MADTP Packet

Now we must detail what must happen when a MADTP packet is received by a host. This description is slanted towards the case of the multicast packet arriving from the originating client at a remote server but the basic principles also hold true for the reverse unicast path from the server to the client. See Figure 6-2 for a flowchart of this operation. The host receiving the MADTP packet first checks if the packet is part of an outstanding transaction from which some packets have already been retrieved. It does this by scanning the list of structures (see Figure 6-3) relating to partially retrieved transactions for one which contains the same source IP address, port and TID as the received packet carries.

If a match on all three of these fields is found, the packet sequence number is checked against the records in the linked list of missing packet sequence numbers held by the receiver. If the packet is a retransmission of a packet that has already been found, it is discarded and the receiver makes a note of the time that the packet was received in the outstanding transaction data structure. This time stamp is used later to determine if a partially completed transaction has timed out and thus its associated data structures should be destroyed in the receiver.

If the sequence number of the received packet falls within one of the ranges of missing packets, the simple MADTP user identification field is extracted from the packet header and matched against the corresponding fields in the outstanding transaction structure. If it fails to match, the packet is silently discarded and the time stamp is not updated.

If the user identification information is valid, the data from the packet is inserted into the request buffer for this transaction at the correct point (as determined using the sequence number and the payload length) and the time stamp in the outstanding transaction structure is updated. The receiver then has to modify the missing packet sequences linked which is attached to the outstanding transaction structure to indicate that the packet has been received.

If it finds that this was the last packet needed to complete the transaction sequence, it is able to process the request. This can be done either synchronously or asynchronously. In the

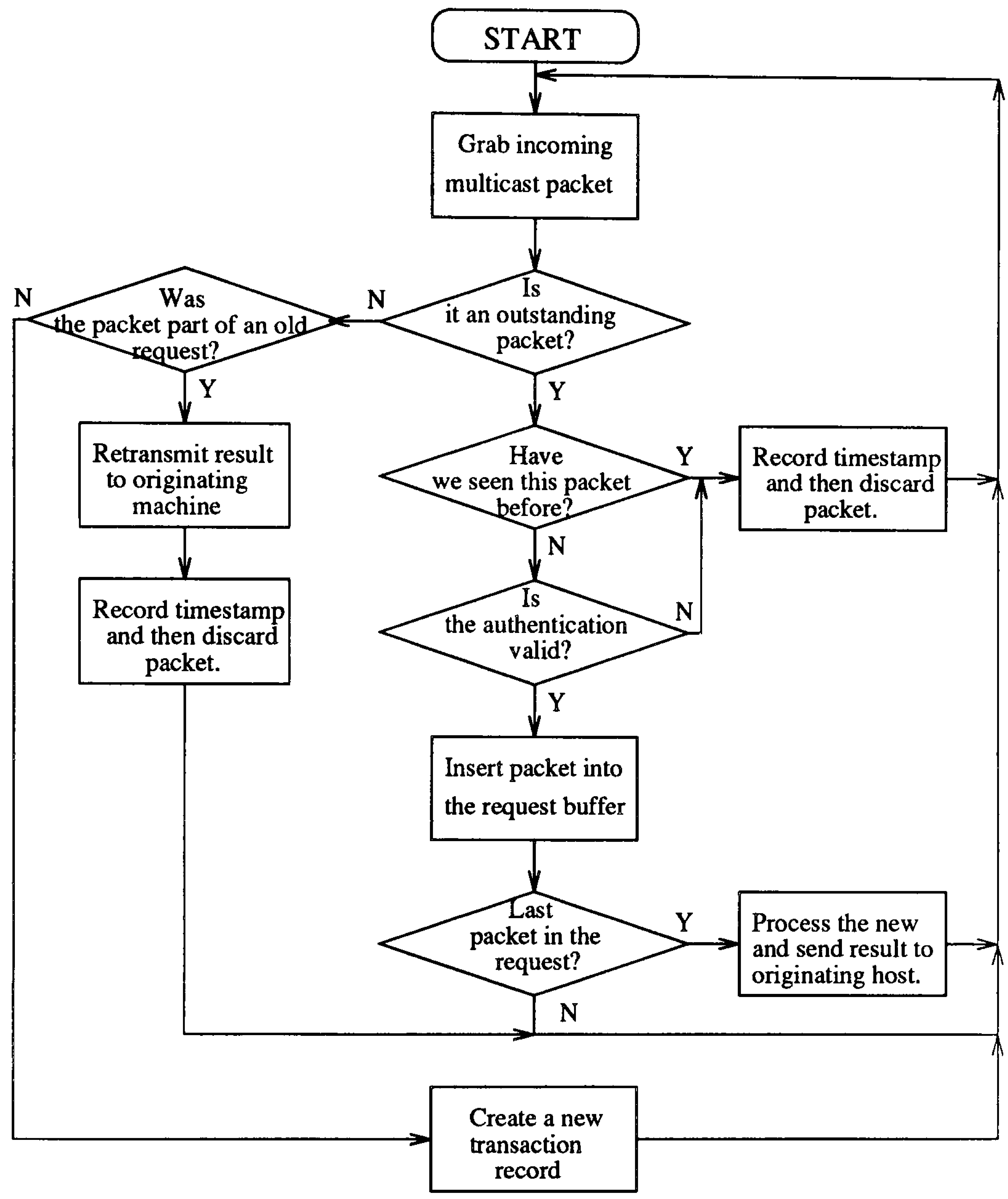


Figure 6-2: Flowchart of MADTP receiver operation

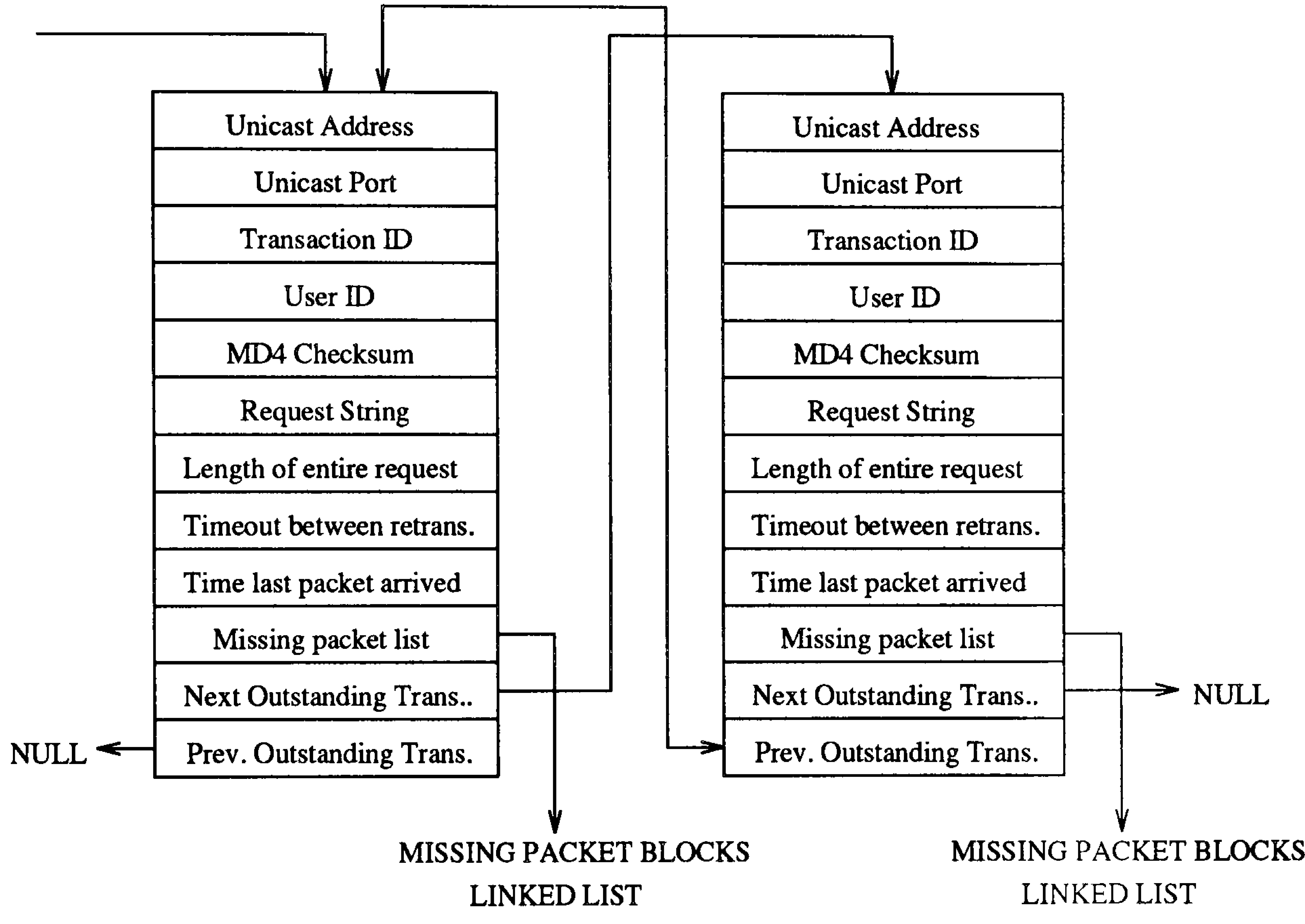


Figure 6-3: Format of outstanding MADTP transaction data structures

former case, the process simply evaluates the request, returns the response and makes a note of the result in a temporary file for possible retransmissions before going on to process any other incoming packets. In the later case, the packet handling process forks off a child process to handle the evaluation of the exported MLbRPC. Before it does this however it creates a new structure to indicate that the transaction is currently being processed and the removes the outstanding transaction record. It also creates a temporary filename that the child uses to cache the result in case it needs to be retransmitted.

If the packet received does not belong to a partially complete outstanding transaction, the receiver checks its IP address, port number, TID and UID against the records held in the linked lists of transactions that the server is either currently processing or for which it has cached answers. If a match occurs, the main server processing code changes the time stamp held in the matched record and forks off a new child process to retransmit the result. The result itself is stored in a temporary file created by the parent before the child is forked.

The last condition is that the packet is neither a member of an existing partially complete transaction nor a fully received transaction which is being processed or for which a result has been cached. In this case the packet must be the first packet to be received in a new transaction (note that this is not necessarily the same thing as the first packet in the transaction's packet sequence; dropped packets or reordering could easily result in a later packet in the sequence being received first). The receiver handles this case by first checking the simple user authentication information in the packet header. It does this by using the IP address and UID to search a database of authorised <IP,UID> pairs. If there is no match, the packet is discarded. This technique will not stop a malicious attacker posing as someone else. However it is quick to do and will allow the server to rapidly discard packets from users that are not malicious but who are not authorised to export code to that particular machine. The next chapter discusses the security issues surrounding LbRPC mechanisms in more depth.

If the simple authentication is successful, a request buffer is allocated whose size is determined by the total transaction length in packets, supplied in the MADTP packet header, multiplied by the MADTP payload length. The receiver then creates a new outstanding transaction structure and adds it to a linked list of outstanding transactions. This structure has the IP address and port number of originating host recorded in it, along with the UID, TID, the

Operation	*Locally Executed*	*Exported (MADTP)*
Null Op.	69μs	134058μs
Simple `set`	970μs	154746μs
Simple `regsub`	7512μs	187412μs
Complex `regsub`	1513698μs	192308μs
Simple maths function	107042583μs	4924219μs

Table 6.1: Performance comparison of non-exported test case TCL code fragments against the same code running over both raw multicast UDP and MADTP transports.

packet sequence length and a pointer to the request buffer. The receiver also starts to build the list of missing packet sequence numbers and attaches this list to the outstanding transaction structure.

6.5 Performance

This section details some performance measurements made using the prototype MADTP implementation on a variety of platforms. Table 6.1 shows the results of transmitting the test code samples used in Chapter 5 over the new transport protocol. The machines used where the SGI Indies used before and once again the network and machines were otherwise unused. The MADTP prototype used MD4 hash functions to check the integrity of the data as will be outlined in the next chapter, but there was no encryption used.

This could almost be viewed as a worst case scenario for the server; it is getting a large number of requests being delivered continuously and the contents would actually fit in one UDP packet. The former factor means that the MADTP based MLbRPC server has to fit a lot of internal house keeping into a short space of time. The latter means that there are an additional two packets sent per request. In practice, requests may consume far more space than the simple test cases and so the additional packet sent using MADTP used to contain security information will be a relatively low overhead. However it still shows that MADTP based transactions can still offer a performance win over the same code run locally on the 386 PC.

One should also remember that the UDP based MLbRPC implementation outlined in the previous section did not fork off any child processes to handle the execution of the exported code

in the remote machines. The MADTP version does do this as it allows multiple transactions to be handled by a remote server simultaneously. It also uses signals to inform the main server code that child has completed execution. The downside to forking off child processes in this way is that the fork operation can be fairly lengthy on many platforms and can result in expensive context switching. On the SGI Indy machines to which the code was exported in the above tests a call to `fork()` appears to take around 8ms and this is without having a separate signal handler to catch the falling children. Some operating systems offer the concept of lightwieght threads and it may be appropriate to use these to reduce the overhead on the server.

6.6 Discussion and Conclusions

In this chapter the need for a multicast transport protocol suited to the needs of MLbRPC has been outlined, along with some reasons why existing MTPs may not be suitable. An MTP designed specifically to support MLbRPC using large groups of servers over WAN links has then been presented, a prototype version of which has been used in conjunction with the MLbRPC prototype described in Chapter 5.

It should be noted that the implementation of MADTP described in the previous section is, like the implementations of the UDP based MLbRPC mechanism detailed in the previous chapter, merely a proof of concept prototype. There is much that could be done to improve the performance of the implementation in a production system. Firstly, the prototype MADTP uses `SOCK_DGRAM` sockets which result in MADTP packets being encapsulated in UDP packets. If MADTP were implemented using `SOCK_RAW` sockets, this overhead could be dispensed with, albeit with less potential portability of code. The forking of child processes could also be replaced by multiple lightweight threads sharing the same address space as the main server code. Lastly the use of intermediate files in the file system may impact on performance and so the use of shared memory or memory mapped files could help.

Even so, it must be realised that there will naturally be some overhead incurred for the extra facilities a sequenced multicast transport protocol provides. The ability to send multi-packet sequences is important for MLbRPC as the size of a single packet constrains the content of the transaction far too much. MADTP differs from many other MTPs by being oriented

towards a single sender transmitting a unidirection multicast stream out to a potentially large and dynamically varying group of listening servers that use unicast communication to return results. As such it appears to be a better match for the needs of MLbRPC than other, more generalised, MTPs.

Multicast transport protocols are still very much under research and development by the network community and, as with unicast transport protocols, it is unlikely that a single protocol will be ideal for all applications. It has only been in the last couple of years that the global MBONE virtual multicast network has permitted large scale trials of any multicast software and lessons are still being learnt about the problems and features of this style of communication. The protocol described here provides another data point in the multicast transport protocol design space and has shown how such protocols can be tailored to the needs of specific higher layer mechanisms.

Chapter 7

Security Considerations

7.1 Introduction

Up to this point, it has been implicitly assumed that the client machine making the LbRPC request will be allowed to export its code to the chosen remote LbRPC server and have it executed. This will not, of course, always be the case. As the basic premise of the LbRPC mechanism is that it allows the client machines to export arbitrary sections of code to remote machines for execution, it is likely that system administrators would feel somewhat uneasy at the prospect of their machines being flooded with LbRPC requests from all over the network. If the LbRPC mechanism is to be used in a production environment, it must support a means of allowing clients to authenticate themselves to remote servers. The remote LbRPC servers are then free to turn away requests from clients that they know nothing about.

Likewise, a client may wish to ensure that the remote host that processed its LbRPC request is a known, valid server, especially if the client picked a specific host in order to access some special resource. WANs such as the Internet offer ample opportunities for malicious attackers to intercept a client's request and return a fake and possibly incorrect result before a real remote LbRPC server ever gets to see it. Thus the client must be given the option of verifying the identity of a remote LbRPC server that has processed a request.

As the networks are so open to allowing abuse of data in transit, the application programmer or end user may also wish to encrypt the exported code and data, and likewise the returned results, so that they are of no use to anyone monitoring the transaction. Only the authorised

clients and servers should be able to encrypt and decrypt the exported modules and results. This is more likely to be desired in the military and commercial worlds than in academia, as in the former environments the code and data being exported may be able to give a potential enemy or competitor vital information about the organisation sending the request.

The purpose of this chapter is to briefly look at how these security concerns can be addressed in the LbRPC mechanism. In his original thesis, Partridge suggested that developments in email security may simply be applied to LbRPC. We will see that, although this is broadly true, it must be done with care if we are not to start incurring multiple network transits simply to provide a secure exportation mechanism. First the various types of security concerns in an LbRPC mechanism are examined in more depth, and the security facilities provided in email systems are overviewed. Then mechanisms for authenticating the requests and responses from clients and servers are then detailed, followed by an investigation into alternatives for encrypting the requests and responses. Lastly, the more general outstanding issues within network security and that may also affect LbRPC are discussed.

7.2 Security Classifications

Security in computer systems really encompasses three major topics; confidentiality, integrity, and availability. The first of these, confidentiality, is covered by security mechanisms that protect the privacy of individuals and systems. In a network based computing environment confidentiality is concerned with preventing a malicious outsider from being able to view the contents of a request or response as it passes through the network between a client and a remote server. In some cases, especially in sensitive military and government systems, security mechanisms designed to protect confidentiality may go as far as disguising what client is talking to what server when view from outside. Traffic analysis can be of great use to some attackers, especially if they can then use it to form the basis of further attacks, such as reducing the availability of service.

Confidentiality is implemented using *encryption* technology. Encryption is the process of converting the source *plaintext* data stream into a *ciphertext* stream that is unreadable to those that do not possess the correct secret *key* to *decrypt* it. It is worth making the distinction

between simple encoding of a data stream and encryption. Encoding of data is simply the translation from one data format to another; it is usually a two way, open algorithm and there is no need for secret keys to decode the resulting data stream. Encoding of data is often used to ensure that a data stream can be safely carried across a transmission medium without loss. Anyone that can intercept an encoded data stream can decode it to gain the original source data. Encryption on the other hand often relies on the secrecy of the key to provide its cryptographic strength; ideally it should not be possible to reconstruct the input plaintext from the ciphertext stream without the secret key. The secret key should not be sent across the network in plaintext form at any point. If a secret key is compromised then the confidentially of any data streams encrypted using that key are also compromised.

The integrity of a system is the assurance that all parts of the system are legitimate and have not been corrupted or replaced by malicious entities. Much of the security functionality found in existing operating systems is designed to ensure that the integrity of the computer system can not be compromised, either accidentally by a legitimate user or malfunctioning application program, or by a malicious attacker.

In a distributed system, integrity checks are needed to ensure that servers and clients are who they claim to be; this is known as *authentication*. When the server has successfully authenticated the client, it can then check what resources the client is *authorised* to use, based on the security policy. It may be that a client that can be successfully authenticated so that the server thinks it knows exactly who it is talking to, may still not be authorised to use the server's facilities and its processing requests can be denied.

Integrity checking in distributed systems is more involved than the integrity mechanisms that a stand alone machine must implement as not only is there a potentially insecure set of network links between the various machines that make up the distributed system, but also the entire system is unlikely to fall under a single administrative domain. As a security system only implements an administrative *security policy*, the entire distributed system can only be as secure as the weakest administrative domain if the domains trust one another. Some integrity mechanisms implemented in more recent network security systems are inherently untrusting of other parts of the distributed system. This inherent distrust limits the damage that a single compromised part of the distributed system can have on the rest of the system. However it

tends to involve an increased communication and/or computational overhead.

Availability of a system is one of the hardest parts of a security system. Some of the "classic" security attacks on complex systems involve what are known as *denial of service* techniques. A common denial of service attack used in the past on stand alone computer systems is the locking out of system administrators from their own machines by repeatly entering incorrect system administration passwords. The operating system's user authentication code would often disable an account after a large number of incorrect logins as a security feature, but in this case it worked to an attacker's advantage by denying service to the real administrator of the machine while the attacker continued to work on the machine. A denial of service attack against a distributed system can be as simple as consuming all the bandwidth available on a network link to a particular host so that it is unable to communicate with other machines on the network.

Replication of network links and remote servers can help limit this type of attack as can the imposition of bandwidth rate limitations in the network. Replication of resource is expensive but it does have the benefit of also providing increased fault tolerance. In fact, one can consider denial of service attacks as artificially induced faults in the system, so any mechanisms designed to improve a systems overall fault tolerance are likely to reduce the harm caused by these types of attacks. When looked at in this light, MLbRPC can be seen to be more resistant to denial of service attacks than the non-multicast LbRPC as an attack on one server is less likely to be noticed because many other servers can still respond. In a non-multicast LbRPC system, flooding the server that a particular client has decided to use could prevent or delay the execution of the client's request on that server. Obviously the single source host is still a vulnerable point of attack even in a multicast environment.

7.3 Securing Electronic Mail

Electronic mail is one of the most widely used facilities in computer networks and its popularity is steadily growing. Unfortunately, in its simplest implementations it can also be one of the most insecure forms of communication available. The widely implemented "Internet RFC822" style mail [38], coupled with the Simple Mail Transfer Protocol (SMTP) [137] and the Internet's current lack of network level encryption, make it ridiculously easy to forge mail and snoop on

mail messages passing between particular hosts. The only tool required is access to a copy of a "telnet" client that can connect to the SMTP port.

As email becomes used for conveying more sensitive information, both personal and commercial, there has been a demand for a facility for proving that a certain individual did send an email message, even if that individual subsequently attempts to deny it. This facility is known as *non-repudiation* of the origin of the email. It requires the ability to digitally "sign" a document in such a way that only one individual could have generated it.

To address some of these problems a number of security mechanisms have been designed for use with email. One of the most comprehensive proposals is known as Privacy Enhanced Mail (PEM) [99, 91, 6, 90]. Specifically, PEM provides a framework to allow email message bodies to be encrypted, to ensure that authenticity of the originator of the message to be determined by the recipient, to guarantee that the message's integrity can not be compromised without detection and to permit non-repudiation of messages by their originator.

Encryption in the PEM architecture can be either symmetric or asymmetric. A symmetric encryption system is one in which both the originator and recipient of a message share the same secret keys in order to encrypt and decrypt the data. Examples of symmetric encryption schemes include the Data Encryption Standard (DES) [46] and LUCIFER. The asymmetric encryption technology uses separate keys to encrypt and decrypt the ciphertext messages. An example of an asymmetric encryption scheme is the RSA Public Key Cryptosystem [56]. The originator of the email message encrypts the plaintext in the public key of the recipient, that, as its name suggests, is publically available. However, once the ciphertext is generated, the public key can not then decrypt it. Only the recipient can do that by using the associated private key, which is kept secret.

PEM permits a number of actual encryption algorithms to be used to encrypt messages, demanding only that a basic symmetric encryption mechanism be available in all implementations for key distribution. The asymmetric encryption option is also used to form the basis of the non-repudiation service. This service involves a third party host that is a *certification authority* trusted by both the sender and recipient of a message. Using private key, the certification authority signs a certificate that already contains a string signed by the originator in its private key. The certificate is returned to the originating site which can then embed it in other

email messages. The recipients of these messages can then contact the certification authority to check that the certificate in the message was indeed generated by the sender of this message and the certificate was signed by the trusted third party.

The specifications for PEM, and for similar systems, have been publicly available for some time now. However take up of the mechanism has been limited. Part of the reason for this apparent lack of interest in PEM stems from the fact that the public key technology developed by RSA Inc. has been licensed to the Internet community for use in PEM in the United States only and can not be exported elsewhere. Encryption technology is treated as munitions by the United States Government and there are strict controls on the export of encryption devices and software, even to "friendly" countries[1]. Other digital signature mechanisms for email that have shown slightly wider deployment, such as Phil Zimmerman's Pretty Good Privacy (PGP) package, are either on dubious legal ground due to infringements of United States export regulations or have been developed outside of the United States[2].

This political restriction on effective cryptographic systems is likely to be more of a burden on a mechanism such as LbRPC which is unlikely to be at all popular unless it can be well secured. At the time of writing, it still appears unlikely that countries such as the United States will relax their cryptographic export laws to permit the successful development of anything but the most primitive or crippled cryptographic systems[3].

7.4 Security in Distributed Systems

Before looking at how security issues can be handled in LbRPC systems, it is worth looking at the security mechanisms that have been implemented in previous distributed computer systems. It is surprising how many distributed systems have very relaxed security policies. For example, of the eight different traditional RPC mechanisms overviewed by Tay and Ananda [176], only

[1]It should be noted that the United States, although the most visible and influential country to restrict cryptographic trade and use is not alone. Closer to home, France also heavily restricts the use of cryptographic material

[2]The United States munitions controls over cryptography appear to apply to imports of cryptographic materials into the United States as well as exports to the rest of the world. This means that a cryptographic system developed outside of the United States would be illegal to import into the country and may even be illegal to use

[3]For instance, the United States Government seems quite happy to allow the use of 40 bit RSA public key cryptography in products destined for export, even though this level of security has been shown to be relatively easy to break. The current key length that is considered secure is 2048 bits or larger.

three provide a security mechanism. This seemingly lax attitude to security may be the result of a number of factors.

Firstly many previous distributed systems come from the academic research community. This community historically tends to avoid tough security features; openness is sometimes seen as a desirable feature. Also distributed systems developed as part of research projects are often concentrating on investigating specific features of distributed computation and security is either forgotten or tacked on as an afterthought. Security mechanisms usually require some extra computational overhead that can affect performance comparisons between different distributed computing mechanisms.

A last reason for lax security is that before personal computers became prevalent, it was difficult for an attacker to gain access to messages as they passed along the network. This approach can still be seen in the IP feature of "secure" ports numbers with numbers below 1024. While this implementation was vaguely reasonable when the machines that used it were under tight, trusted administrative control, it is now completely defeated by having untrusted users double as systems administrators. These users have super user status on their own machine and can start any service on these supposedly secure ports. The personal computer also gives attackers access to a cheap and powerful network "sniffer"; they can often monitor traffic passing between other hosts on the network. This is especially true on shared media networks such as Ethernets and FDDI rings. Therefore "secure" port ranges and network traffic can no longer be considered to be at all secure.

The rapid spread of personal computers and network access in to the hands of untrusted users and malicious attackers has lead to the development of a number of security oriented distributed systems and network operating system extensions. These include the S/Key [65, 64] and Kerberos [95] systems that have both been widely deployed in a variety of environments.

S/Key is a mechanism that allows users to log in to and execute applications on remote machines without having to send a plaintext password across a potentially insecure network link. S/Key utilises a nested set of one way functions to convert the user's unchanging plaintext password into an authentication bit string known as the "one-time password". The one-time password that is generated for the user is useless if observed by an attacker performing passive eavesdropping as it can not be reused at a later date.

When a user wishes to access a remote service, the client first passes the username to the remote server. The server responds with the number of times the one-way function should be applied and the remote host's seed. The seed allows a user to reuse the same secret key across a number of hosts whilst still generating different sequences of one-time passwords for every host. The client then takes the user's secret password, concatenates the host's seed and applies the one-way function to it to give an 8 byte result. This result is then feed back into the one-way function again; this process is repeated the number of times specified by the remote server. The final result is returned to the remote server which then applies the one-way function to it and then checks the result against a prestored version. If they differ the authentication failed; otherwise the user is authenticated and the stored password on the remote server is updated with the one-time password that the user has just used. The next time the same user tries to contact this server, the number of iterations demanded will be reduced by one.

S/Key is secure against passive attack as any attacker would need to be able to successfully reverse the one-way function at least once to give him the one-time password that would be generated with one less iteration than the one he captured. S/Key is only used at the start of a potentially long lived session and so the two network round trips that are incurred are negligible compared to the total session lifetime. It does have the disadvantage that eventually the user must generate a new stored password on the remote host because the number of one-way functions applied by the client is reduced by one each time the remote host is contacted.

The Kerberos system developed at MIT as part of the Athena Project in 1986 [95, 113] is somewhat more powerful than S/Key. Kerberos is intended for environments where the users are potentially untrustworthy and have full access to at least one machine one the network. Kerberos makes use of the concept of limited lifetime tickets. The tickets are cryptographically secured tokens that are provided to legitimate users by the system to permit them to use a specific resource or set of resources. The whole security of a particular Kerberos system, or *realm*, depends only upon the security of a limited number of administration and authentication servers. These machines have to be kept under tight control and are usually located in a physically secure environment as well as being well protected against electronic attacks.

The administrative and authentication servers have access to a database containing the name, private key and expiration date of a *principle*. A principle is not just a user; it can also

be a network service. The difference between the administrative and authentication servers is that the former allows read and write access to the Kerberos database whereas the latter only provides read access. It is the authentication server that authenticates principles in the Kerberos systems and issues them with tickets. Authenticators and tickets are both based on symmetric private key encryption technology [4]. The difference between authenticators and tickets is that a client can generate authenticators itself and they are only used once. The tickets are generated by the authentication server in response to a successful decrypted authenticator and can be used a number of times before it expires.

Kerberos works well in a local area environment but has not really spread into usage over WAN links. Although this is partly due to the difficulty of getting different administrative realms to allow cross realm authentication, it is also related to the fact that a full Kerberos authentication takes five network transits. Although this introduces little overhead in a local area environment, it can affect performance over WAN links with high latencies, especially as authentication must be obtained for access to every separate server that the user wishes to use.

7.5 Implementing Security in LbRPC

The LbRPC mechanism is similar to the email environment from a security point of view in that messages will be received from an initially untrusted party and the recipient has to be able to determine the authenticity and authorisation attached to each message. It is also likely that should LbRPC be widely deployed, there will be a desire amongst users to encrypt the outgoing code and data and the returned results in an LbRPC transaction. Lastly, a single LbRPC request can be sent to multiple hosts simultaneously using the Multicast IP based MLbRPC mechanism proposed in Chapter 5.

However, there is one notable difference between the email and LbRPC environments. That difference is the amount of time that the system has to perform authentication. In email systems, the user is accustomed to long delays between transmission and reception of a message and the return of a reply, if any. There are many periods in the delivery process that the mail system has plenty of time to make multiple network transits to remote third party public key

[4]The symmetric private key cryptosystem actually used Kerberos is the DES in CBC mode just as in PEM

servers in order to support authentication and encryption mechanisms. The LbRPC system does not have this luxury; as its purpose is to minimise the effect of network transits over WANs, the querying of remote servers for authentication information upon receipt of a message is unacceptable. The clients and servers must be able to perform their authentication using cached information that can be initialised "out of band".

In the LbRPC mechanism, the first security matter to attend to is authentication. There are two sides to this aspect of system security; the client must be able to authenticate itself to the remote server when it sends its request, and the server must authenticate itself to the client when it returns a result. The server does not wish to waste time and endanger itself processing code for unauthorised clients and the client does not wish to accept responses from potentially malicious remote servers. Both the client and the server can use similar mechanisms to perform this task. Ideally, the authentication information should be self contained once a message has been received by a server or client. The receiver should not need to communicate with third parties in order to authenticate the message as this would incur extra network transits.

7.5.1 Simple Authentication

A "straw man" proposal for message authentication is to simply place a user identifier in the header of the LbRPC packets. The user identifier should represent the user that is executing the client application or that the remote server runs under. It could be as simple as the UNIX UID field from the user's password file entry. When linked to the originating host of the packet, it then identifies a unique user in the network as the UID is unique on a single host. The receiver merely has to look up the authorisation attached to a particular <host address,user identifier> pair to see if it should continue to process the message.

This method has the advantages of being easy to implement and quick for both the sender and receiver to process. However it is not at all secure; any user on a machine can extract the UIDs for all other users from the password file[5] and even if an attacker does not have access to the originating host, it is still possible to obtain the information by observing packets in transit. This is a known deficiency in using static information for authentication; it is routinely used by hackers in attacking `telnet` or `rlogin` sessions that send the same user password in plaintext

[5]Even on systems that employ shadow password files.

every time a remote login is performed for example. It is a good example of why letting the message carry all the authentication information is a bad idea.

7.5.2 Asymmetrical Digital Signatures

It is therefore desirable to employ a mechanism that does not send the same information on every message and that also makes use of a secret piece of information known only to the legitimate sender and receiver. One means of doing this would be to use a digital signature [51]. This digital signature could be used to either sign an entire message or sign each individual packet that is used to transport that message between the sender and receiver. If the signature is used to sign each packet, the receiver can throw away any packets it fails to authenticate successfully without any further processing. However, digital signatures involve the application of encryption algorithms as was discussed in the email section above and these are typically computationally expensive.

As the server is likely be processing a large number of incoming packets this is clearly not desirable. The alternative of processing the digital signature after all the packets in a message have been received reduces this overhead slightly, although the time taken to decrypt the digital signature is still likely to be proportional to the length of the message itself. The downside to this is that it opens the server up to a denial of service attack where a malicious third party continually bombards the server with large requests that it has to buffer and check the digital signatures on. Even if all the digital signature checks fail, the attacker has still managed to tie up a lot of computational resources on the server. However, most machines connected to WANs today already offer ample ways for an attacker to execute denial of service attacks that this may not be a major worry.

The digital signature itself could take many forms. It could use a method similar to PEM's digital signatures and allow non-repudiation but this potentially involves the use of a third party. Non-repudiation in the LbRPC mechanism could be desirable in some circumstances. For example, if a client discovered that a result that had been returned by a remote server turned out to be incorrect, it may be useful if the server is unable to deny all responsibility for generating it.

To facilitate digital signatures which are capable of supporting non-repudiation, the LbRPC

mechanism has to therefore use an encryption system to generate the signature which supports the use of asymmetric keys. Symmetric keys can not be used because the symmetric key is a shared secret between the client and the server and so the server would be capable of generating fake requests that appeared to have been encrypted by the client. It must also provide a trusted third party that will vouch for both the client and server. In LbRPC, accesses to this third party would probably only be done after the transaction had been completed and if there was a dispute between the client and the server's administrators. Allowing non-repudiation to take place at run time is not really an option in LbRPC as it will involve multiple additional round-trips over the network.

A possible mechanism for using asymmetric public key technology to encrypt LbRPC and also provide digital signatures is now outlined. Assume that a client c wishes to send an LbRPC request message m_p to a remote server s. The client has to be able to encrypt the message and some form of identifying digital signature, send the request to the server, have the server decrypt the message and correctly authenticate the client, process the message and then return a similarly encrypted response to the client with the server's own signature in it.

The digital signature that the client initially uses has to be a small value that it can encrypt in manner that no other machine or user is capable of doing and also which is unique for this transaction. In this way, the client can not claim either that someone else generated the signature or reused a signature that they had seen in the past. One way to generate this small value is to use a hashing or checksum algorithm on the entire request message that is being sent.

There are a number of possible candidates for the checksum algorithm. Simple polynomial checksums, as used in the lower layers of the network to guard against packet corruption, are quick to implement and can detect simple changes to the message. However, it has been shown that it can be relatively straightforward for the skilled attacker to generate a fake message that has the same polynomial checksum as the original legitimate message. A better alternative is a checksum algorithm specifically designed to make it hard to generate a forged message that has the same checksum as a legitimate message.

Examples of this latter class of checksums are the RSA Message Digest (MD) family[89, 145, 146] and the US Government's Secure Hash Algorithm (SHA-1) [156]. The MD series

generate a message digest that is a fixed length "fingerprint" from an arbitrary length input data stream. The fingerprint is a 128 bit number in all three algorithms and all are designed to make it computationally expensive to generate two input data streams that result in the same fingerprint. The algorithms are reasonably fast[6]. The MD2 algorithm is much slower than the MD4 and MD5 algorithms and uses a smaller block size in its internal computation. The MD4 and MD5 algorithms are similar, with MD5 being a slightly stronger algorithm but slightly slower. MD4, although "it is "at the edge" in terms of risking successful cryptanalytic attack"[146], has not yet been successfully compromised.

However, the MD family of algorithms have been shown to be possibly too slow to use at high line speeds [177] and are more computationally expensive than some other proposed hashing functions. As the hashed value computed in this application is later itself encrypted, a faster, simpler algorithm may be more appropriate. The use of secure digest checksums in high performance environments has only just begun to be investigated in depth and so the choice of checksum algorithm will be left open ended here.

Assuming that a suitable hash function H is used, the client uses it on the message:

$$h_{out} = H(m_p) \tag{7.1}$$

This hash value is then encrypted by the client. Denoting asymmetric encryption using the key key as E_{asm}^{key} and the corresponding decryption as D_{asm}^{key}, the client uses its public asymmetric key X_c to give:

$$E_{asm}^{X_c}(h_{out}) \tag{7.2}$$

The client can then generate the message m_{out} to be sent to the server by encrypting the plaintext message m_p, and the result from step 7.2 using the remote servers public key X_s:

$$m_{out} = E_{asm}^{X_s}(m_p, E_{asm}^{X_c}(h_{out})) \tag{7.3}$$

When the remote server receives m_{out}, it first decrypts it using its own private key K_s:

[6]Even on a relatively low performance workstation such as a Sun IPX, MD5 can generate fingerprints from data stream at a speed in excess of 10Mbps

$$
\begin{aligned}
D_{asm}^{K_s}(m_{out}) &= D_{asm}^{K_s}(E_{asm}^{X_s}(m_p, E_{asm}^{K_c}(h_{out}))) \\
&= m_p, E_{asm}^{X_c}(h_{out}) \qquad (7.4)
\end{aligned}
$$

The server must now recompute the hash value found in Equation 7.2 and compare it to:

$$D_{asm}^{K_c}(E_{asm}^{K_c}(h_{out})) \qquad (7.5)$$

If they match, the server knows that the client was the originator of the message. If they don't match the server can discard the message and possibly log an attempted attack.

Assuming the authentication of the client was successful, the server can then take the plaintext message m_p and process it to generate a result m_r to be returned to the client. The process it undertakes is a mirror image of the client. First it generates the hash value of the result message:

$$h_{back} = H(m_r) \qquad (7.6)$$

It then encrypts this value using its private key to give:

$$E_{asm}^{X_s}(h_{back}) \qquad (7.7)$$

This is then encrypted along with the message itself using the client's public key X_c:

$$m_{back} = E_{asm}^{X_c}(m_r, E_{asm}^{X_s}(h_{back})) \qquad (7.8)$$

Finally the server sends m_{back} back to the client. The client can then decrypt this using its private key K_c to recover the reply and the encrypted hash value. The latter of these can then by decrypted using the server's public key that the client holds and checked against a locally generated version of 7.6

Note that for this method to work, the clients and servers must have some way of indicating to the other party which user they are acting on behalf of, so that the correct keys can be

retrieved to decrypt the message. The simple straw-man proposal of including a UID in the packet headers could be used safely in this case as if a copy of the UID from a previously seen packet was used in a rogue request, the secret keys would still be unknown and so the receiver would not be able to decrypt the digital signature on the received message.

One problem with the method describe so far is that most asymmetric key encryption mechanisms are quite slow. Asymmetric key systems such as the RSA public key system rely on the fact that it is computationally difficult to generate the private key from public one or from the ciphertext. They do this by utilising things such as products of large prime numbers in their algorithms which would require an attacker to factor out the large primes in order to compromise the system. Unfortunately they also tend to be somewhat computationally intensive when encrypting and decrypting the data stream with legitimate keys. As the length of the time required to encrypt or decrypt a message is usually related to its length, they are best used on small messages. An LbRPC request could potentially be quite large, upto several megabytes if it has to carry a lot of data with it.

An alternative to encrypting the entire message with the asymmetric keys would be to only encrypt a small block of data with it that could then be used to ensure the integrity of the rest of the message. There are two possible ways of doing this. One would be to only encrypt the checksum results for the messages. Only the holder of the correct corresponding asymmetric key would be able to decrypt the checksum and use it to check the integrity of the rest of the message. Providing a fake checksum encrypted in the wrong key or altering any part of the message would allow the receiver to detect a forged or altered message.

Although checksums are much quicker to generate and check over the large body of the message than the asymmetric encryption algorithms are, they obviously leave the actual message in plaintext format. This may be acceptable in situations where good performance is required and the system must only ensure that an attacker cannot inject fake messages without needing to ensure that he cannot read the contents of legitimate ones. For situations in which the contents of the messages must be obscured from passive observers, it may be desirable to still encrypt the message contents.

7.5.3 Dual Encryption

Luckily, symmetric key cryptosystems tend to be much faster than asymmetric ones whilst still providing a high degree of cryptographic strength. An example of this would be the DES used in it Cipher Block Chaining (CBC) mode[46]. CBC mode makes cryptanalysis of the cipher text more difficult by feeding back the results of the encryption of one block of the plaintext into the encryption of the next. This is desirable in this application not only because it makes the ciphertext more difficult to break without the secret symmetric key but also means that tampering with any part of the encrypted ciphertext will cause repercussions through the rest of the ciphertext.

One way to utilise this better performance would be to have the originator of the message encrypt it using a symmetric key encryption algorithm with a randomly generated key. This key could then itself be encrypted using the asymmetric key known to the sender. Only legitimate receivers that possess the corresponding asymmetric key will be able to decrypt the encrypted public key and then use it to decrypt the message itself. An attacker who either changed the encrypted public key or altered the ciphertext of the message would be detected by the receiver as the message would no longer be able to be decrypted successfully.

Thus in the above description, assume that the client uses a symmetric encryption algorithm E_{sym}^{key} with the key P_c to encrypt m_p after the hash value has been computed. Then instead of using Equation 7.3 to generate the message to be sent to the server, the client uses:

$$m_{out} = E_{asm}^{X_s}(P_c, E_{asm}^{X_c}(h_{out})), E_{sym}^{P_c}(m_p) \tag{7.9}$$

The server can decrypt the first part of m_{out} using its private asymmetric key to give it the symmetric key P_c that it can then use to recover the plaintext m_p from the message. When the server has generated a result m_r it can encrypt that using a symmetric key P_s and return it to the client in a similar manner. Note that the symmetric keys P_c and P_s become shared secrets between the server and the client. However both parties are free to generate new random keys for each transaction so this is not a problem.

7.5.4 The Multicast Environment

An additional problem occurs with both the purely asymmetric and dual asymmetric/symmetric encryption methods described above if the client and server are in a multicast environment and are using MLbRPC. The problem is that the multiple servers that may be able to answer a request must share a single private asymmetric key in order to decrypt the client's m_{out} message but the client will receive no reliable indication of exactly which server(s) actually generated replies.

To overcome this problem it is necessary for the MLbRPC servers to be able to supply the client with some information that will uniquely identify it. One way to do this would be for each server could use a different asymmetric key pair to return the reply to the client, the private side of which only that particular server would know. The client can then match the unicast IP address returned by the server and retrieve the correct public asymmetric key for that server. The servers should also encrypt the computed hash value of the result, $H(m_r)$, using the personal asymmetric private keys rather than the K_s that shared with other servers in the multicast group. As only one server should have a particular personal asymmetric private key, the client can be sure that it knows the server that the result was returned from.

7.5.5 Performance

To give the application programmer or user flexibility in the strength of security and performance required, the LbRPC mechanism should really permit either a checksum such as MD4 or a symmetric key encryption system such as DES in CBC mode to be used. It would be up to the application programmer or user to determine whether the extra security of symmetric encryption would be worth the performance loss against a non encrypting checksum. For the really paranoid, the system could even allow both to be used, so that the checksum was generated from the ciphertext and this could be checked before attempting to decrypt the message.

One should note however that using either encryption of any kind or many secure checksum algorithms will impose an additional time overhead on each exported request. Using current encryption technology on existing hardware, these additional delays are roughly comparable to the latency experienced over the network. As machines get faster, the encryption and decryption of messages should get faster as well. However even on a 1000 MIPS machine as envisaged by

Partridge in his original work, this may well mean that LbRPC requests containing a megabyte of code and data might several tens of milliseconds to process by the security software at each end. This is equivalent to adding another long haul network transit to the LbRPC request.

7.6 Payment

With the increased emphasis on commercial usage of the Internet in recent years, it is worthwhile considering how LbRPC transactions can be securely accounted and charged for. Although this is not likely to be of great interest in much of the academic community, there is always the prospect of commercial entities allowing remote users to make use of their computational resoures using LbRPC on a "pay-as-you-play" basis.

There are two basic mechanisms that can be used to charge a user for making use of a remote service over the network. Firstly, the user may be forced to set up an account with the service provider in advance and then will be billed periodically for any use he makes of the service. This charging model is similar to the billing mechanism employed by various traditional public utilities. It can make use of the authentication information described in the previous section and need carry no additional information with the requests.

The alternative method of charging the user is to have each request carry the information required to bill the user immediately. This is the form of billing that is finding favour in commercial enhancements to the World Wide Web [144, 110]. In this case the authentication information sent in the request either sends the user's credit card information or some form of electronic cash. The transmission of encrypted credit card details is likely to be the most widely deployed in the short term but electronic cash offers a number of advantages over credit cards. Firstly, service providers may feel obliged to check the credit worthiness of the card before they will start to satisfy the user's request. The time required to provide this check may be acceptable for processing requests in the World Wide Web where the end user will expect credit card validation to take a few seconds. However, in the LbRPC environment, this is an unacceptable delay.

Secondly, the credit cards may impose an artificial lower bound on the cost of the the transaction which may be very much higher than the actual cost required to handle the transaction.

Credit cards are not the ideal means of paying for very low value transactions. Digital cash comes into its own here because it is possible for it to pay for services in small fractions of a penny.

Some of the digital cash mechanisms that have been proposed are designed to work without having to query a central service. This allows them to be handled more rapidly. In the original proposals this speed was often needed because the digital cash was intended to pay for packet routing in the network on a hop by hop basis. It may also prove useful in LbRPC where the low overhead allows transactions to be turned round much faster at the remote server. Also, as the digital cash value attached to a request is set by the originating client, the user will not be subject to unscrupulous server administrators attempting to charge more than the user is prepared to pay to satisfy a request.

One further problem that is encountered with charging is what to do in the case of a multicast transaction. The sender of the request in a MLbRPC system will not know in advance how many remote servers an outgoing request will reach and how many of them will reply. If a user is prepared to pay a set maximum amount to handle a request, how can this payment be split up amongst the various servers that may answer his request? This problem faces not only MLbRPC but any system that uses multicast IP and requires payments to be made for services.

One scenario is that the user may only have to pay for the replies that are actually utilised. This is preferable from the user's point of view as money is only expended for requests that are actually used by the local application. However, it is far from ideal from the servers' points of view as they may not get paid even if they expend considerable resources to satisfy a request. It also means that the servers are forced to trust that the user will actually pay someone once the results have been returned. If the per request cost of LbRPC processing on a server is relatively low then the loss of the occasional payment may be acceptable and server administrators can black list regularly non-paying users. However, if the per request cost is high, the server administration will want to claim payment on as many transactions as possible.

The opposite to having the user only pay for the replies that are used is to have the server bill the user for all requests that the server processes even if the user does not make use of the returned result. This ensures that the servers are recompensed for the effort they use in fulfilling a user's request. Unfortunately in the MLbRPC multicast environment described in

Section 5 the group of servers that might accept the user's request is open ended. The need for authentication and authorisation detailed in the previous sections will serve to limit the number of servers that will actually process the request. This reduction in the number of servers that can actually process the request can be used by the user to limit the maximum number of servers that will bill him and thus the maximum amount of money that a request will cost.

It should also be noted that charging for processing LbRPC requests can be seen as introducing a new variable into the placement decision. The MLbRPC solution to process placement satisfied the single goal of minimising total execution time at the expense of additional network and CPU usage. If this additional network bandwidth and CPU time must be paid for by the user, MLbRPC may no longer be such an attractive solution to the process placement problem. If this is the case, then the placement decision may have to revert to using more computationally intensive algorithms. The cost variable also shatters the transparency of the network to the end-user; the system is likely to be instructed to clear transactions that involve payment of more than a minimal amount over a predefined period with the end user.

7.7 Conclusions

It is obvious that there is a tradeoff between raw performance and the security of a system. Any implementation of security mechanism is an extra set of tasks that must be performed every time an LbRPC transaction is undertaken. It should also be clear now that LbRPC systems are not likely to be deployed on a wide scale until end-to-end security mechanisms can be put in place. From a security point of view, LbRPC is a very dangerous protocol; it allows a remote user to execute an arbitrary segment of code on a server. Systems administrators will need to be convinced that LbRPC gives them at least the same level of security as is afforded to ordinary user shell access on current machines.

As has been shown in this chapter, the security issues in LbRPC are very similar to those faced in the email environment with really only one major difference; the time scales in which the security software must operate are greatly reduced in the LbRPC environment. Security in email systems is free to operate on human time scales and users are likely to put up with a few seconds delay whilst a certificate is checked with a remote third party. The same thing is not possible

in LbRPC; multiple network transits are exactly what are being avoided. The authentication, authorisation and decryption of messages must be achievable by the receiver with no external interaction with third party machines. These restraints also differentiate LbRPC from existing distributed systems security mechanisms that often utilise multiple network transits in order to authenticate a single request.

The proposal outlined in this chapter allows both clients and servers to authenticate one another and ensures the integrity of the message. Privacy concerns are addressed by permitting the messages to be encrypted when they are transmitted over the network. Authentication and cryptographic keys have to be held in databases on both clients and servers. This information is not updated at all by the LbRPC protocol; existing secure key distribution technologies can be used.

It is important to realise that the security of an entire system is often only as good as its weakest link. In the case of the proposed security mechanism for LbRPC, if a client is compromised by other means and an attacker gains access to the client's key database, the attacker can then fool all remote servers into executing requests on his behalf and will be able to decrypt their replies. If the server implements a restricted execution environment for the exported code in the same way that Safe-TCL does for enabled active email the consequences of this on the server can be minimised. Due to the use of asymmetric keys, the attacker will not be able to pose as a server unless he actually compromises the machine the server is running on.

Unfortunately, although the security mechanism required for LbRPC appears to be technically feasible, the existing political situation makes its deployment difficult. The treatment of cryptographic technology as munitions with tight export restrictions will impinge upon the legal use of secure LbRPC over international links, at least for commercial and academic users. As LbRPC is specially designed for use over the sort of links found between continents and international collaborators wishing to share expensive computational or specialist resources are likely to be its main users, this political restriction could seriously affect the viability of LbRPC in the real world.

Chapter 8

Discussion and Conclusions

8.1 Introduction

This thesis has described an in depth evaluation of the feasibility of the Late-binding RPC mechanism. Although the basic premise of reducing the number of network transits that a distributed computation has to endure is still valid, the actual LbRPC mechanism proposed by Partridge appears to have some fundamental limitations. These limitations have been described in detail in the previous chapters. In this chapter they will be briefly reiterated and their severity assessed. Some discussion will also be made of the lessons to be learnt from this piece of research and how it may be extended in the future.

8.2 Review of limitations

8.2.1 Intermediate Code Representations

The first of the limitations in the original proposal of LbRPC is the inability of any current language to provide a sufficient range of representational power to allow it to be used efficiently as an intermediate representation for the exported code. LbRPC as described by Partridge appears to have fallen into the "UNCOL trap"; it is trying to be far too general. Many efforts in the past have shown the folly of attempting to over generalise the ability of a system. One is left with either a wooly specification that appears at first glance to be general enough but that is impossible to implement, or a working implementation that has to compromise its full

generality.

Although far more is known about the design and usage of programming languages today than when the first UNCOL effort was proposed nearly forty years ago, programming languages are still are fertile area of research and development. It would appear to be impossible to produce a fully general intermediate representation as it would have to be able to support programming paradigms that have not yet been invented.

Indeed, Partridge's idea of being able to export arbitrary portions of code may in itself be less than desirable. As the work in the field of enabled active electronic mail has shown there is a desire on the part of both users and administrators to limit what exported code is capable of doing. By limiting the generality of the exported code, the dangers it poses to systems are also limited.

This raises the question of just how general, or how limited, the exported code in an LbRPC mechanism will need to be. There is likely to be no firm answer to this; it will vary based on the needs and abilities of the end users, the application programmers, the system administrators and the LbRPC implementors. However it is likely that there are a number of different intermediate representations will be chosen to support different tasks. For example, there may be an intermediate representation optimised for text processing and another which is oriented more towards scientific mathematics. It may even mean that the intermediate representation merely becomes a restricted subset of an existing language. This would remove the need to translate the source language into the intermediate format, would allow the application programmer to write code in a language familiar to him and would ease the difficulty of implementing the LbRPC as existing compilers and interpreters can be used.

Not all remote servers would have to handle all possible intermediate code representations; many would probably only handle one or two. This in itself makes sense as different host architectures provide differing support for the various programming paradigms in use today. It would be up to the application programmer to ensure that he selected the appropriate representation for the problem that he is attempting to solve. Another advantage of using multiple representations is that the decision as to whether the exported code should be interpreted on the remote host, or compiled into machine code before executing it, can be done on a case by case basis. Some applications are likely to benefit more from compilation that others. For

example, a mathematical algorithm may be encoded in a relatively compact program that is relatively quick to compile compared to the amount of time the actual execution would take if it must operate over a large body of data. As was mentioned in Chapter 3 it may even be desirable for the application programmer to be able to specify whether code should be interpreted or compiled based on external knowledge about the likely execution and compilation times of the exported code resulting from experience with the algorithms used or profiling of previous executions.

Unlike intermediate exportable code formats, external data representations have been shown to be a fairly stable and well developed area of distributed systems. Using techniques such as deep copying and arbitrary object sizes, it appears that it is possible to export practically any object between different platforms. LbRPC should be able to simply make use of one of the existing external data representations such as XDR or ASN.1, or even use a string based representation of the exported data as the TCL based prototypes presented in this thesis have done, to handle the data being exported between machines.

8.2.2 Process Placement

This thesis has shown that the placement of the exported code and data in a LbRPC system can make a significant difference to the experienced performance of the distributed application. It has also shown that attempting to retrieve network and host performance measurements at the time the code and data is about to be exported is pointless; the time taken to acquire the necessary performance measurements can be far greater than the actual execution time of the exported code. Some of the measurements may also be difficult or impossible to obtain in an wide area internetwork environment as the paths between distant hosts have rapidly varying bandwidths and latencies due to congestion and packet loss from the large number of other traffic sources.

This thesis has proposed two means of overcoming this problem. The first is to make network and host performance measurement an ongoing process that is performed continually by a single host for a local cluster of machines. The machines in the local cluster then request performance information on desired remote hosts from this monitoring station in order to make the process placement decision. This thesis has suggested that the monitoring station can utilise some

existing network management and performance monitoring tools to acquire the necessary data. However determining the set of remote hosts to query for performance information and even reliably making the process placement using heuristic algorithms are difficult to do because of the scale of the internetwork environment.

The second approach to placing code and data in an LbRPC based system makes use of multicast network layer communications. In this scenario the local host actually makes little or no explicit process placement decisions. The only real decision required is what multicast group the request is sent to; this decision may be statically compiled into the application by the programmer or it may be done at runtime using multicast group announcements. This solution to the placement problem can achieve near optimal results, especially if used over low error rate links and with a local copy of the exported code being executed in parallel with the remotely executing copies.

However these benefits come at the cost of increased overall network and CPU usage and the danger of executing non-idempotent requests using this mechanism. The first two of these drawbacks can be discounted if we assume that there is ample network bandwidth and CPU resources to "waste" in return for good total execution times. The later is more serious and requires that either only idempotent safe code is exported using this mechanism or that the application programmer makes arrangements to implement an external commit protocol. Neither is ideal; one limits the applicability of the LbRPC mechanism and the other incurs additional round trip delays. Even so, MLbRPC appears to be a better means of placing the code and data in a distributed application than the other alternatives considered.

8.2.3 Lower Layer Issues

The choice of the transport layer used to carry the LbRPC requests and responses has been shown to be vital to the performance of the distributed application. If the transport layer uses techniques such as sliding windows or slow-start congestion avoidance that can incur multiple network transits, all the efforts expended at minimising network transits at the higher presentation and application layers will be wasted. Also the performance of the underlying network protocol will affect the overall performance of the distributed application.

This thesis presented a review of the transport protocols available, with special emphasis

on the currently experimental multicast transport protocols. Many of these are designed with long lived connections operating on human interaction timescales in mind. These are not likely to be suitable for use with MLbRPC style mechanisms as the feedback, group management and congestion avoidance mechanisms often require multiple round-trips or knowledge of all the members of the multicast group. Therefore a new multicast transport protocol, MADTP, has been described which provides better support for MLbRPC's needs.

The development of multicast protocols is the subject of much ongoing research and it is not suggested that MADTP should compete with other protocols such as RMP as a general purpose reliable MTP. MADTP is designed to fulfill a niche market; its sequenced, saturation based mechanism was selected to fit in with the needs of MLbRPC and the high performance networking environment it is destined to operate in. It does however provide a useful data point that other MTPs can be compared against.

8.2.4 Security

The security implications of allowing users to execute arbitrary chunks of code on remote machines are wide ranging and serious. This thesis has detailed how existing encryption and authentication technology can be applied to the LbRPC mechanism. The main problems involved with this are that the stronger the security mechanism is, the more resource intensive and slower the processing of the LbRPC transactions become and that there are currently many non-technical obstacles to deploying strong encryption technology.

The non-technical problems are really beyond the scope of this thesis and will hopefully eventually be resolved as a result of the move towards secure electronic commerce on the Internet. However the technical problem of strong encryption techniques being very CPU intensive could well be a serious drawback to the feasibility of LbRPC. The solution proposed here is to provide multiple levels of security, each of which is increasingly secure but that may take more resources to implement. This not only gives the end user or application programmer the option of trading security for performance but also allows servers to rapidly discard incoming requests that fail to achieve even the weakest level of security. This later point is important if MLbRPC becomes widespread and the number of servers that a particular end user is authorised to use is far fewer than the actual number that see the requests.

8.3 Discussion

This thesis has attempted to investigate the feasibility of implementing an LbRPC mechanism for use over networks with high bandwidth delay product links. The generalised system originally proposed does not appear to be possible to implement with existing technology. Specifically, the lack of an intermediate, exportable code format that is capable of expressing the facilities found in the wide range of programming languages that are currently in use appears to be the biggest problem facing a generalised LbRPC mechanism. This is closely followed by the problem of deciding where to place the exported code in a network with many possible remote servers.

The best way to deal with the first of these appears to be to drop the demand for a completely general intermediate exportable code format and instead deploy a number of different exported codes, each of which is targeted at a specific application area. This solution is in keeping with the observed trend in high level application oriented programming languages where there are a number of different languages in common use, each with its own specialised application area that it is suited to best.

The placement decision work described in this thesis has relevance to not only LbRPC but other systems that involve resource discovery and resource utilisation optimisation phases. The main difference between the needs of LbRPC and many of these other situations is that LbRPC imposes much more severe time constraints upon both phases. Resource discovery research that is currently underway in fields such as networked information retrieval typically allow the resource discovery mechanism to take several seconds to return its answer. In extreme cases, resource discovery may be viewed as an ongoing process in these environments running for weeks or months. In the LbRPC environment, both resource discovery and utilisation optimisation should be completed as rapidly as possibly and should really only contribute a small fraction to the total execution time of the exportation mechanism.

The novel multicast based placement mechanism described in Chapter 5 removes the need to perform an explicit resource discovery phase and then pick or estimate the optimal resource to actually use. The use of multicast communications to discover and make requests of multiple remote resources may be useful in applications other than LbRPC. This is a very active field of research at the moment and one which is ripe for further investigation.

Similarly the development of the MADTP transport protocol fills a niche in the multicast transport protocol design space that has hitherto not been addressed. Its provision of a sequenced, semi-reliable multiple packet multicast transport layer for arbitrary sized groups of receivers fulfills the needs of the MLbRPC system, but may also be applicable in other similar systems. As with resource discovery, large scale multicasting is still in its infancy and it is hoped that some of the ideas presented in the MADTP prototype may be taken up by other multicast transport protocols. Further work needs to be performed to assess the scalability of the saturation approach to reliability used in MADTP as the size of the group of responding servers grows and if the error rate of the links is variable.

Overall, this thesis has shown that the platform and source language independent LbRPC mechanism proposed by Partridge has several feasibility problems that will prevent its full implementation. This is not to say that subsets of the LbRPC mechanism should not be deployed; the goal of maximising the amount of work performed on every round trip over a network is still very desirable. LbRPC-like mechanisms targeted at specific applications are likely to prove very useful in building distributed systems over network links with high bandwidth-delay products. It is hoped that the work presented in this thesis can be used as a base from which this further work can be developed to fruition.

Chapter 9

Appendices

9.1 Appendix A: Initial Simulation Results of Late Binding against Traditional RPC

9.1.1 Introduction

This appendix details the results of a number of simulations of late binding and traditional Remote Procedure Calls (RPC). The purpose of these simulations was as a "sanity check" of the concept of LbRPC and as such have much of the detail of network and host loading variations removed from them. The simulations also did not set out to prove that LbRPC would be better than traditional RPC in all settings; for small operations over fast LANs traditional RPC still has a speed advantage. Instead these simulations simply demonstrate that the concept of LbRPC could provide a speed advantage in at least one reasonable scenario and therefore confirm the results of Partridge's thesis [131].

Line errors and buffer overflow errors were not considered in the simulations. The target gigabit WANs that LbRPC is intended to be used on are likely to have very low Bit Error Rates and have forward error correction for those line errors that do occur. Some research has shown that algorithms for buffer sizing exist which can make buffer overflows sufficiently rare that they do not impact greatly upon performance.

The effects of other traffic on the links has also been ignored in these simulations for a number of reasons. Firstly, the general effect of other traffic on network links is to reduce the

available bandwidth to the protocol under study, and so heavily loaded links will appear to just have reduced bandwidth. Bandwidth related effects have been looked at in the simulations. Secondly, it is likely that on future wide-area gigabit networks, the hosts will have the ability to set up virtual connections with the provision for bandwidth guarantees. How this will actually work is still the subject of ongoing research.

The simulation results were also compared to an experiment carried out on a real, unloaded Ethernet at Loughborough over the Christmas period of 1992. In this experiment a remote filesystem was mounted via the RPC based NFS mechanism and a `find(1)` was executed looking for two files named "greek". The filesystem contained 10903 separate searchable entries, requiring 11917 RPC calls to be made to search through the entire filesystem. Other traffic on the network at the time of the experiment was on average around 3 packets per second. The NFS based `find` took on average 96 seconds (averaged over 18 separate runs spread through a 2 hour period). A similar `find` run locally on the machine with exported NFS discs took only 32 seconds on average. These figures were used to produce "sensible" estimates for various parameters in the simulations and to check that the simulation results for traditional RPC matched those measured.

9.1.2 Effect of communications delay on total execution time

In this simulation, the basic communications latency of the communications link was varied, whilst keeping all other parameters constant. For traditional RPC communications, a packet size of 200 bytes was considered, with 11917 requests being sent. The requests and responces were both 1 packet long.

The late-binding simulation exported 55000 bytes of code as this seemed a reasonable assumption for the size of an exported `find(1)` based upon figures from the TDF Facts and Figures report and the size of the BSD NET2 Release source code. The returned results were only 300 bytes long, which also seems reasonable considering that the real experiment that the simulation was being benchmarked against only matched 2 filenames in the filesystem being searched, and it is these names plus some control information that would need to be returned.

The bandwidth for both traditional and late-binding simulations was 10Mbit/s. The communications latency was varied between 0 and 150msec in 1msec increments (figure 9-1).

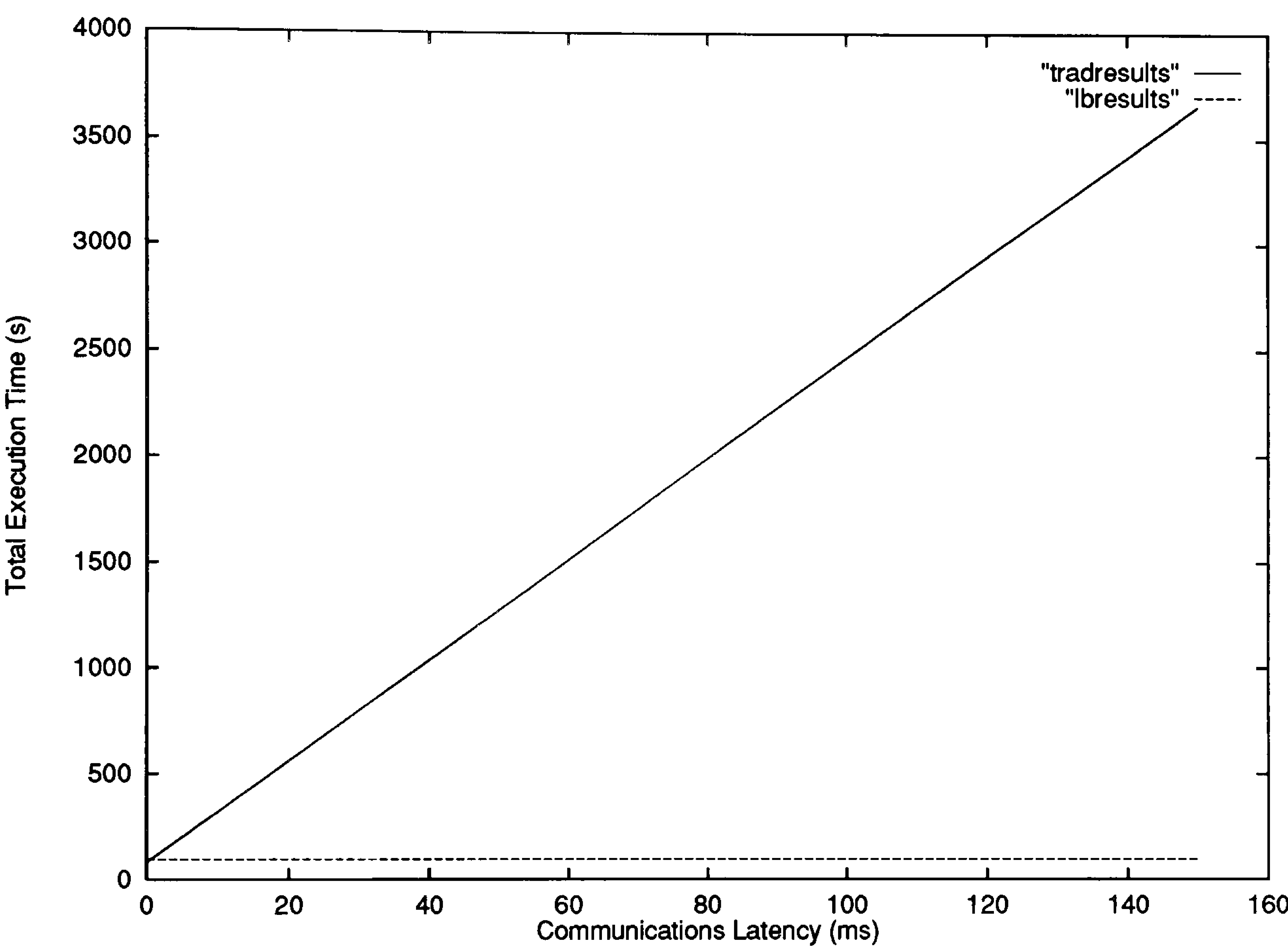


Figure 9-1: Latency against time for a 100Mbit/s throughput

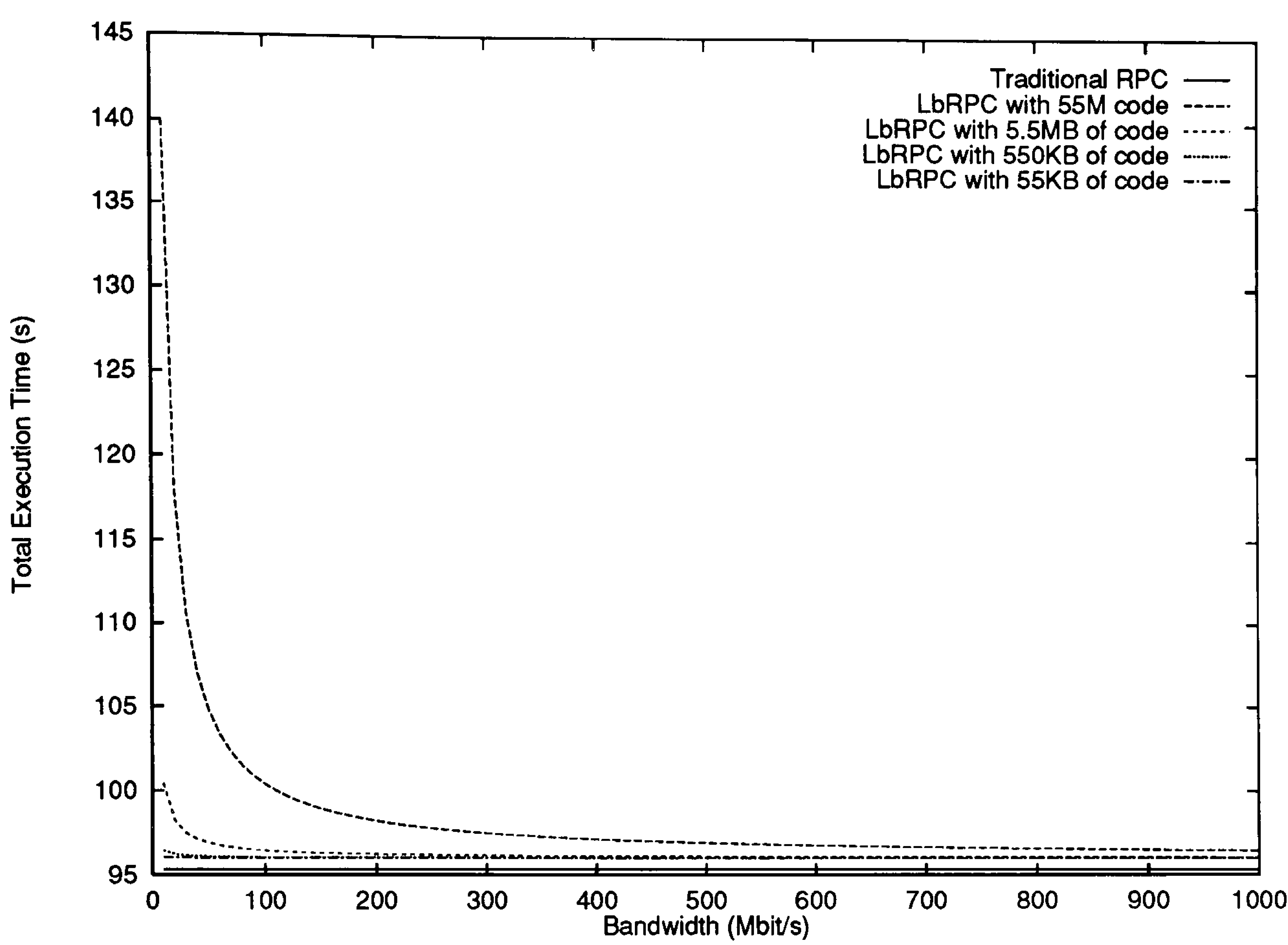


Figure 9-2: Effect of throughput with a fixed latency of 500μsec

9.1.3 Bandwidth Effects on Total Execution Time

The effect of varying the effective bandwidth of the communications link for both traditional and late-binding RPC was studied. The bandwidth was varied between 10Mbit/s and 1Gbit/s in increments of 10Mbit/s. For the traditional RPC simulation, all other parameters were held fixed. However, for late-binding RPC, a number of simulation runs were performed with different exported code sizes (although the size of the returned results was kept constant). Exported code sizes started at 55000 bytes (to compare against the other simulations) and then were multipled by 10 upto 55 million bytes.

The same simulations for both traditional and late-binding RPC have been performed with underlying communications latencies of 500μsec (figure 9-2) and 150msec (figure 9-3) to see how bandwidth and communications delay affect one another.

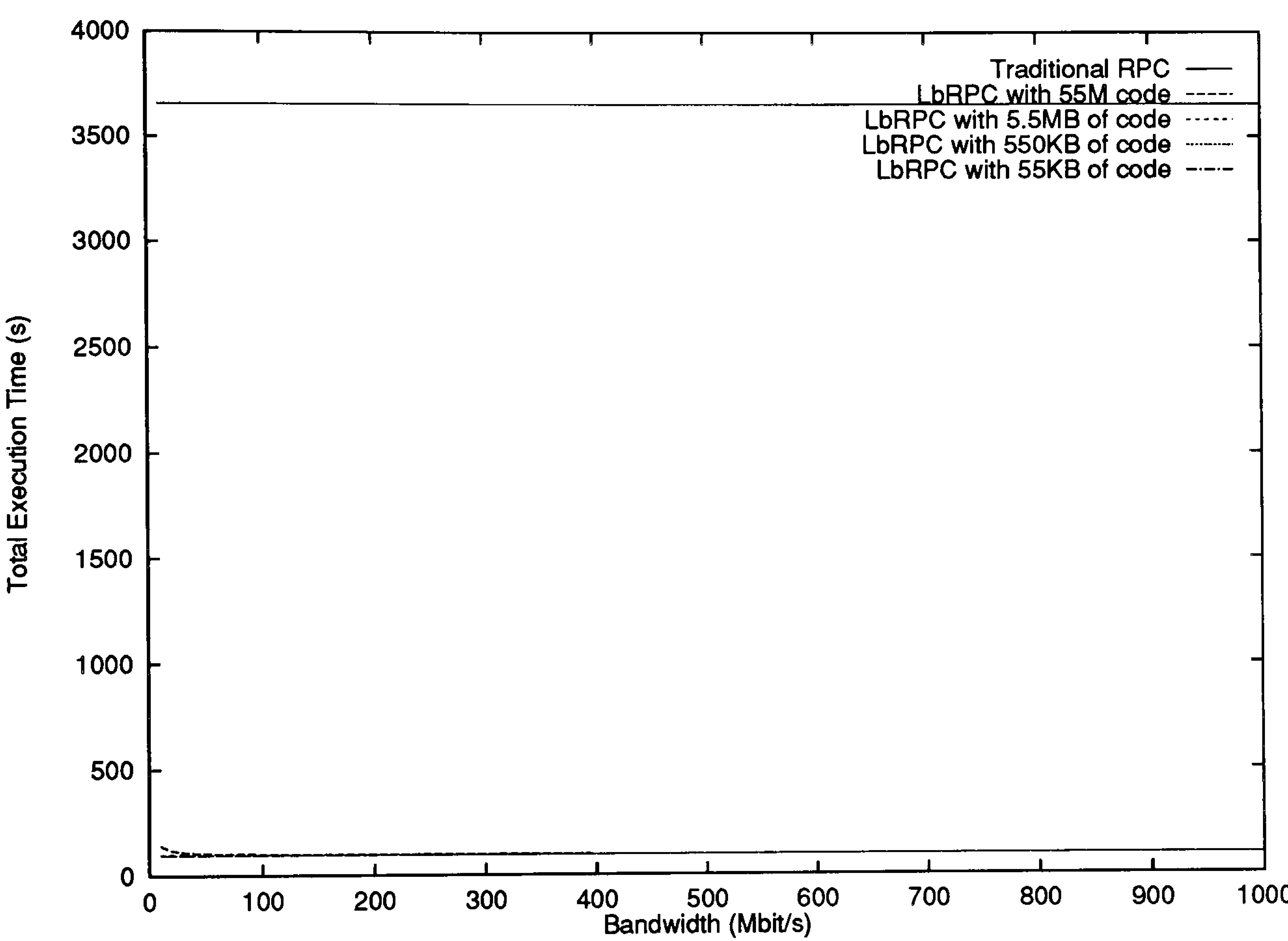


Figure 9-3: Effect of throughput with a fixed latency of 150ms

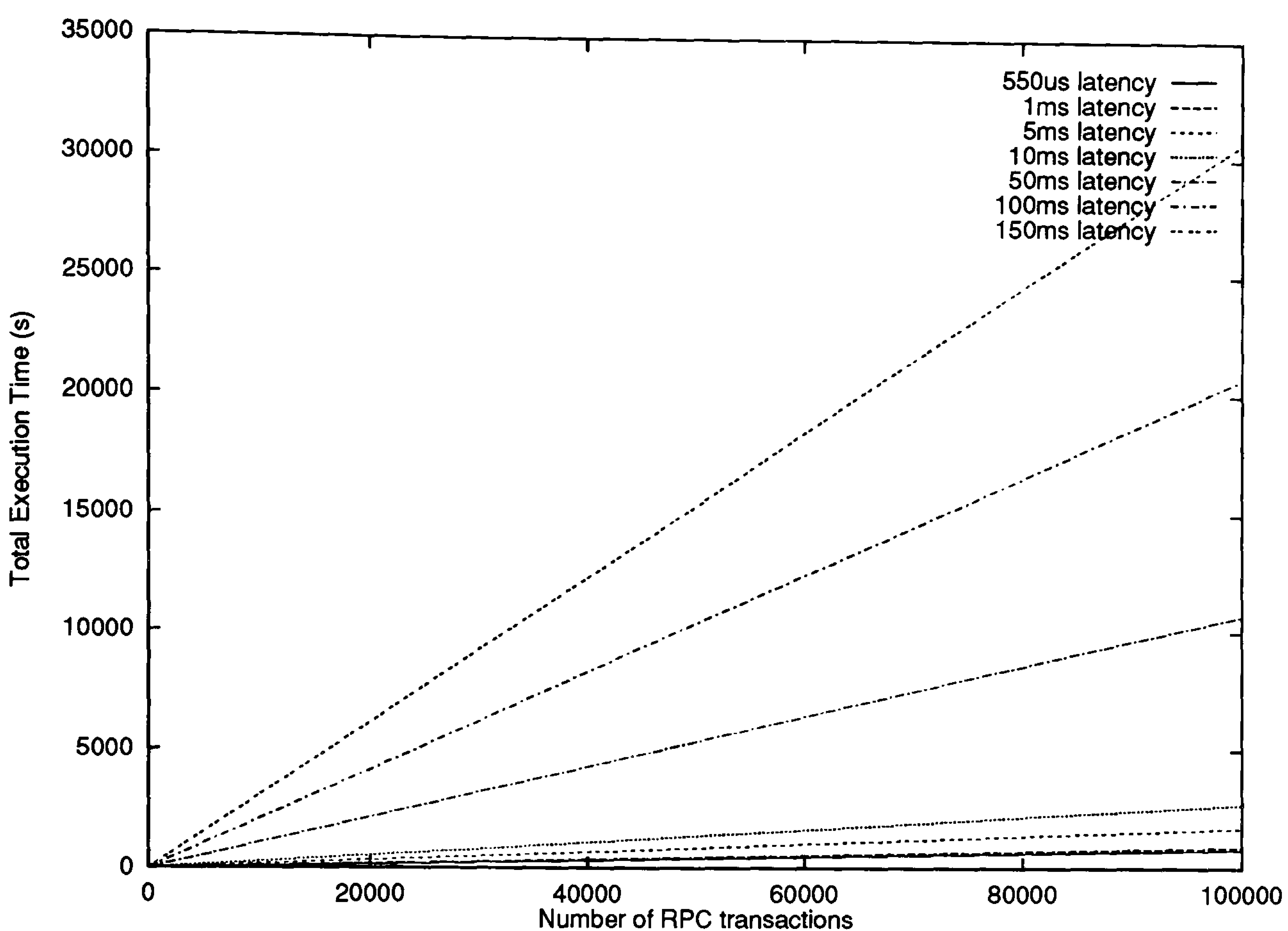


Figure 9-4: Number of RPC transactions against time with different latencies

9.1.4 Cost of number of traditional RPC requests

In his thesis, Partridge showed that the number of network transits must be minimised in order to develop acceptably efficient distibuted systems over networks with large latencies. This simulation shows this result to be correct (figure 9-4); only traditional RPC transactions were simulated, with the number of requests varied between 0 and 100000. The simulation was run with a number of different communications link latency delays; 500 μsec, 1ms, 5msec, 10msec, 50msec, 100msec and 150msec. It shows graphically that as the communications delay is increased, the number of network transits made has an increasing impact upon the total execution time.

9.1.5 Conclusions

The graphs of the results clearly show that late-binding RPC can offer considerable performance improvements over traditional RPC in applications which require large numbers of traditional

RPC requests to be made or which operate over a large latency. The breakeven point where late-binding RPC appears to be a performance win depends upon all the factors of bandwidth, latency, install time, remote execution time and number of traditional RPC calls which would be required to perform the task in hand.

Chapter 10

Glossary

ANDF The *A*rchitecture *N*eutral *D*istribution *F*ormat. Designed by the Defense Research Agency in the UK, this system has been adopted by the Open Software Foundation as a potential mechanism to permit the creation of cross-platfrom "shrink-wrapped" packages. As it is designed to be both source language and target hardware independent to some extent, it is a possible candidate as an intermediate code representation for LbRPC.

ASN.1 *A*bstract *S*yntax *N*otation One. A formal abstract syntax language used defined by the ISO and used in a variety of network systems for host independent representation of data structures. ASN.1 must use a transfer syntax such as BER or PER to unambigously transfer its data structures from host to host.

ATM *A*synchronous *T*ransfer *M*ode. A widely used method of cell based networking that is becoming increasingly popular with vendors of high performance network transmission systems. ATM is capable of scaling from bandwidths of a few megabits per second up to a gigabit per second or more. However its small cell size and leanings towards connection oriented communications can make it somewhat inefficient for computer networks.

BER The *B*asic *E*ncoding *R*ules. A standardised ASN.1 transfer syntax.

BER *B*it *E*rror *R*ate. This is a measure of the number of erroneous bits that one would see on a network link. It is measured in bits/s. Small BERs are good as it means that packets are less likely to be damaged in transit. Fibre optic gigabit WANs have very low BERs

and consequently allow higher level protocols to be optimised for the case of a successful, rather than corrupted, transmission.

Broadcast The ability to transmit a message over a network to all connected hosts. Broadcast is widely used on LANs where the number of hosts is small but is hardly ever used in WAN environments where the number of hosts listening can be much bigger.

DES The *D*ata *E*ncryption *S*tandard. A widely used block cipher originally developed by IBM. It is reasonably fast and fairly secure against cryptanalysis, especially when used in it Cipher Block Chaining mode.

DSM *D*istributed *S*hared *M*emory. A distributed computing mechanism whereby memory used by processes within the system can be spread throughout the available host processors. DSM has been widely used in LAN based distributed systems but the high latency and heterogeneous nature of WANs tends to limit its appeal for WADCS.

ICMP The *I*nternet *C*ontrol *M*essage *P*rotocol is the error and control message protocol used by the Internet protocol family. It is used to report errors and for network diagnostics and management. It is an unreliable datagram protocol layered above IP.

IETF The *I*nternet *E*ngineering *T*ask *F*orce. This is the organisation responsible for the design and implementation of the Internet protocol family. It is a loose body formed from a number of working groups and has recently started to work more closely with the ISO and the other telecommunications standards bodies.

IP The *I*nternet *P*rotocol. IP is the network layer protocol that forms the foundation of the global Internet.

ISO *I*nternational *S*tandards *O*rganisation. This is a multinational standards body, part of which deals with computer and telecommunications standardisation. It is known for the OSI Seven Layer Reference Model and a large number of protocols that fit into its framework. Unlike the IETF, ISO does not require protocols to be independently implemented and interoperated before standards are finalised. This has resulted in a number of standard protocols that are difficult, if not impossible, to implement and a correspondingly lower usage than the Internet protocol family experiences.

Kerberos A security system devised at MIT which allows users and programs to securely communicate with untrusted hosts and programs. It is widely used on LANs but is not widely deployed over the public Internet, mainly due to the effects of high latency links and the need for administrative domains to cooperate.

LAN A *L*ocal *A*rea *N*etwork. Although the definition of "local" varies it is usually taken to mean a network that covers a limited geographical area such as a room, a building or a campus. Historically LANs have had higher bandwidths and lower BERs than WANs but the introduction of fibre optic based gigabit WANs is removing these differences.

LbRPC *L*ate-*b*inding *R*emote *P*rocedure *C*all. A distributed computing mechanism originally proposed by Craig Partridge that replaces the simple argument passing of traditional RPC with the ability to export both code and data to remote machines. The advatnage of LbRPC over traditional RPC is that in WAN environments it can potentially cut down on the number of network transits made and so reduce the effect of RTDs on the distributed computation.

MADTP *M*ulticast *A*synchronous *D*atagram *T*ransport *P*rotocol. An experimental MTP described in this thesis that was specifically designed to be used with MLbRPC. It is a success oriented protocol that uses saturation methods to overcome any packet loss that does occur.

MBONE The *M*ulticast Back*bone*; a virtual multicast network overlaid on the physical topology of the global Internet. Original devised for multicasting audio from IETF sessions it is now almost a production service used to support a wide variety of multicast based wide area communication mechanisms.

MD2/4/5 The *M*essage *D*igests (MD2, MD4 and MD5) are cryptographic hashing algorithms that can generate a secure checksum from a stream of data. The versions differ in their speed and level of security.

MIB *M*anagement *I*nformation *B*ase is a set of data structures that are used by SNMP. MIBs are defined for particular devices or services and provide the common ground for different elements of the SNMP software to make queries and respond correctly.

MLbRPC *M*ulticast *L*ate-*b*inding *R*emote *P*rocedure *C*all. The novel mechanism described in this thesis for obviating the need to make placement decisions in LbRPC systems by multicasting the requests out to a group of servers in a WADCS.

MTP *M*ulticast *T*ransport *P*rotocol. The set of protocols at the OSI Transport Layer that utilise multicast facilities provided by the lower layers to provide group based communication.

Multicast The facility to send a single message from one host to a number of selected hosts in the network that form a multicast group.

NREN The *N*ational *R*esearch and *E*ducation *N*etwork. The US Government initiative to develop a national high performance communication infrastructure that fostered research and development of wide area gigabit networks.

OSI *O*pen *S*ystems *I*nterconnection. The International Standards Organisation's framework to allow interworking between arbitrary systems. It is well known for its seven layer reference model and a variety of (sometime baroque) ISO protocols that adher to the model.

PEM *P*rivacy *E*nhanced *M*ail is a specification for a secure email service built on top of the existing Internet based mail systems. PEM makes use of encryption and digital signatures to provide its service. The model it uses offers a number of options which may be useful in providing a secure LbRPC mechanism.

PER The *P*acked *E*ncoding *R*ules. An ASN.1 transfer syntax that is more compact than the BER.

RPC A *R*emote *P*rocedure *C*all is means of invoking a procedure stored on another machine with a set of arguments and then, optionally, having any results returned. It provides a procedure call abstraction of the network service that is familiar to application programmers. Traditional RPC mechanisms are implemented by nearly all distributed systems but suffer when used over high latency WAN links as the amount of work the fixed procedures that are stored on the remote host do is usually quite small.

RTD *R*ound *T*rip *D*elay. The time taken for a message to pass from one host in a network to another and then back again. It is usually very small in LANs but in WANs can be measured in hundreds or thousands of milliseconds.

S/Key This is a one-time password mechanism that uses a number of iterations of the MD4 cryptographic hash function to hide the plaintext password. The S/Key hash that is transmitted over the network is useless to passive observer for use in replay attacks as the remote host keeps track of the number of iterations of the MD4 hash that have been used so far.

SNMP The *S*imple *N*etwork *M*anagement *P*rotocol. An Internet standard for providing diagnostics, reporting and error detection from remote hosts and services.

TCL-DP The *T*ool *C*ommand *L*anguage with *D*istributed *P*rocessing extensions. An embedded command and scripting language with additional commands added to permit network access and distributed computing. TCL and TCL-DP have been used in this research programme as the basis for a number of prototype systems due to the easy availability of the source code, platform independence and ease of adding new extensions.

TCP *T*ransmission *C*ontrol *P*rotocol. A reliable unicast streaming transport protocol built on top of the IP network layer.

TTL *T*ime *T*o *L*ive. In unicast communications, the TTL field of the IP packet prevents infinite routing loops by specifying a maximum hop count. It has a similar use in multicast IP MBONE where it is used to limit the spread of multicast packets to certain regions.

UDP The *U*ser *D*atagram *P*rotocol is an unreliable datagram transport protocol used on top of the IP network layer. It is used for both unicast and multicast communications.

UNCOL *UN*iversal *C*ompiler *O*riented *L*anguage. UNCOL was an early effort to produce a platform and problem oriented language independant representation for computer programs. It hit upon a number of problems and it was not possible to implement the originally proposed system.

Unicast The ability to transmit a message between a single pair of hosts on a network.

WADCS A *W*ide *A*rea *D*istributed *C*omputing *S*ystem. The coordinated use of computational engines and resources spread over a large geographic area to run application process. The ability to make use of a variety specialised resources such as supercomputers and large database engines in a number of organisations allows a WADCS to tackle problems that a single machine or a LAN based distributed system could not cope with.

WAN *W*ide *A*rea *N*etwork. A WAN is large network spread over a wide geographical area that can contain many hundreds, thousands or even millions of host machines. Due to the long distances between hosts, the RTDs experienced can be large and so latency sensitive protocols are often avoided. WANs have traditionally had a lower bandwidth and higher BER than LANs but the use of fibre optic technology is rapidly changing these features.

XDR The e*X*ternal *D*ata *R*epresentation is a presentation layer protocol devised by Sun Microsystems for use in their Network File Systems and Open Network Computing RPC systems. It has also been widely used in other systems in the Internet. Although it does not offer the same representational power as ASN.1 it is popular because it is much easier to implement fully and the implementations tend to be much faster.

XTP The *X*press *T*ransfer *P*rotocol is a transport level protocol designed from the ground up for high performance communication links and implementation in hardware. It uses bidirectional virtual circuits between end systems and offers a multicast communications mechanism. The multicast communications mode can either have no error detect or flow control, a go-back-N system or selective retransmission. However it does not appear to offer good scaling to very large multicast groups as the transmitter has to maintain state information for a large number of receivers.

Bibliography

[1] F. Anklesaria, M. McCahill, P. Lindner, D. Johnson, D. Torrey, and B. Alberti. The Internet Gopher protocol (a distributed document search and retrieval protocol). Technical Report RFC 1436, IETF Network Working Group, March 1993. Available as <ftp://ftp.ds.internic.net/rfc/rfc1436.txt>.

[2] ARPA Computing Systems Technology Office. *ARPA High Performance Computing and Communications Symposium*, Alexandria, Virginia, USA, March 1994. Available as <URL:http://ftp.arpa.mil/Symposium94/Abstracts.html>.

[3] Y. Artsy and R. Finkel. Designing a process migration facility: The Charlotte experience. *Computer*, 22(9):47–58, September 1989.

[4] G. Attardi and M. Gaspari. Multilanguage interoperability. Technical Report UBLCS-93-18, Laboratory for Computer Science, University of Bologna, Italy, July 1993. Available as <ftp://ftp.cs.unibo.it/pub/TR/UBLCS/MultiLangInterop.ps.Z>.

[5] Joshua S. Auerbach, Arthur P. Goldberg, Ajei S. Gopal, Mark T. Kennedy, and James R. Russell. Concert/c: A language for distributed programming. In *Winter USENIX 1994 Technical Conference*. USENIX, January 1994.

[6] D. Balenson. Privacy enhancement for internet electronic mail: Part iii: Algorithms, modes, and identifiers. Technical Report RFC 1423, IETF Network Working Group, February 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc1423.txt.

[7] Joel Bartlett. SCHEME– >C a portable Scheme-to-C compiler. Technical Report WRL-TR-88.1, DEC Western Research Laboratory, 1989.

[8] Tim Berners-Lee. Universal resource identifiers in WWW. Technical Report RFC 1630, IETF Network Working Group, June 1994. Available as <URL:ftp://ds.internic.net/rfc/rfc1630.txt>.

[9] Tim Berners-Lee, Robert Cailliau, Ari Luotonen, Henrik Frystyk Nielson, and Arthur Secret. The World Wide Web. *Communications of the ACM*, 37(8):76–82, August 1994.

[10] Karl Berry. web2c. Available as <URL:ftp://ftp.th-darmstadt.de/pub/tex/src/web2c/web2c.tar.Z>.

[11] K P Birman and T A Joseph. On communication support for fault tolerant process groups. Technical Report RFC 992, IETF Network Working Group, November 1986. Available as <URL:ftp://ds.internic.net/rfc/rfc992.txt>.

[12] Andrew Birrell and Bruce Nelson. Implementing remote procedure calls. *ACM Trans. Computer Systems*, 2(1):39–59, February 1984.

[13] William J. Bolosky, Fitzgerald Robert P, and Michael L. Scott. Simple but effective techniques for NUMA memory management. In *12th ACM Symposium on Operating Systems Principles (SOSP), Lichfield, AZ, USA*, pages 19–31, December 1989.

[14] William J. Bolosky and Michael L. Scott. A trace-based comparison of shared memory mutliprocessors using optimal off-line analysis. Technical report, University of Rochester, Rochester, NY 14627-0226, USA, November 1991.

[15] William J. Bolosky, Michael L. Scott, Robert P. Fitzgerald, Robert J. Fowler, and Alan L. Cox. NUMA policies and their relation to memory architecture. In *4th International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 212–221, April 1991.

[16] N. Borenstein and N. Freed. MIME (Multipurpose Internet Mail Extensions). Technical Report RFC 1341, IETF Network Working Group, June 1992. Available as <URL:ftp://ds.internic.net/rfc/rfc1341.txt>.

[17] N. Borenstein and N. Freed. MIME (multipurpise internet mail extensions) part one: Mechanisms for specifying and describing the format of internet message bodies. Tech-

nical Report RFC 1521, IETF Network Working Group, September 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc1521.txt>.

[18] Nathaniel S. Borenstein. Email with a mind of its own: The safe-tcl language for enabled mail. In *Proceedings of ULPAA 94*, 1994. Available as part of <URL:ftp.ics.uci.edu/mrose/safe-tcl/safe-tcl.tar.Z>.

[19] Klaus Bothe and Christian Horn. Übersetzung zwischen höheren programmiersprachen:eine lösung des UNCOL-problems? *Angwandte Informatik*, page 283, September 1989.

[20] C. Mic Bowman, Peter B. Danzig, Darren R. Hardy, Udi Manber, and Michael F. Schwartz. Harvest: A scalable, customizable discovery and access system. Technical Report CU-CS-732-94, University of Colorado, Boulder, Colorado, USA, July 1994. Available as <URL:ftp://cs.colorado.edu/pub/techreports/schwartz/Harvest.FullTR.txt.Z>.

[21] S. Bradner. The recommendation for the IP Next Generation protocol. Technical Report RFC 1752, IETF Network Working Group, January 1995. Available as <URL:ftp://ds.internic.net/rfc/rfc1752.txt>.

[22] H-W. Braun and Y. Rekhter. Advancing the NSFNET routing architecture. Technical Report RFC 1222, IETF Network Working Group, May 1991.

[23] T.P. Brisco. DNS support for load balancing. Technical Report draft-ietf-dns-lb-00, IETF Network Working Group, March 1994. Work in progress.

[24] C Burdorf and J Marti. Load balancing strategies for time warp on multi-user workstations. *Computer Journal*, 36(2):168–176, 1993.

[25] J. Case, M. Fedor, M. Schoffstall, and J. Davin. A simple network management protocol (SNMP). Technical Report RFC 1157, IETF Network Working Group, May 1990.

[26] John Cavallini. National challenge applications. In *ARPA High Performance Computing and Communications Symposium*. ARPA, March 1994.

[27] D. R. Cheriton. VMTP: A transport protocol for the next generation of systems. In *SICOMM'86 Conference, Stowe VT (USA)*. ACM, August 1986.

[28] David Cheriton. VMTP: Versatile message transaction protocol, protocol specification. Technical Report RFC 1045, IETF Network Working Group, February 1988. Available as <URL:ftp://ds.internic.net/rfc/rfc1045.txt>.

[29] D.R. Cheriton. The V kernel: A software base for distributed systems. *IEEE Software*, 1(2):19–42, April 1984.

[30] G. Chesson. XTP design. In *Procedings IFIP WG6/WG6.4 International Workshop on Protocols for High Speed Networks*, May 1989.

[31] G. Chesson. The evolution of XTP. In *Proc. 3rd IFIP Conference on High Speed Networking, Berlin*, March 1991.

[32] D. Clark, V. Jocabson, J. Romkey, and M. Salwen. An analysis of TCP processing overhead. *IEEE Communications Magazine*, 27(6):23–29, June 1989.

[33] Melvin E. Conway. Proposal for an UNCOL. *Communications of the ACM*, 1(3), March 1958.

[34] Bob Cooper, Les Clyne, Bob Day, Kevin Hoadley, John Dyer, David J. Pullinger, and Chris Adie. Network news: SuperJANET special edition, November 1993.

[35] R. Cooper. An introduction to SMDS. *Network News*, November 1993. Available as <URL:http://www.ukerna.ac.uk/NetworkNews40/SMDS/SMDS.html>.

[36] Robert Cooper. SuperJANET. *Computer Networks and ISDN Systems*, 26(3):269–274, November 1993.

[37] Digital Equipment Corporation, Hewlett-Packard Company, HyperDesk Corporation, NCR Corporation, Object Design Inc., and SunSoft Inc. The Common Object Request Broker: Architecture and Specification. Technical Report OMG Document Number 91.12.1, Revision 1.1, Object Management Group, December 1991.

[38] David H. Crocker. Standard for the format of ARPA Internet text messages. Technical Report RFC 822, IETF Network Working Group, February 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc822.txt.

[39] Jon Crowcroft, Zheng Wang, and Ian Wakeman. A simple TCP extension to achieve reliablle 1 to many multicast. Technical report, Department of Computer Science, University College London, Gower Street, London WC1E 6BT, United Kingdom, March 1992. Available as <URL:ftp://cs.ucl.ac.uk/darpa/tcp-multicast.lp>.

[40] Ian Currie. Possible ANDF/TDF extensions, 1993. This is a discussion document generated as part of the Esprit project "E.P. 6062 OMI/GLUE" intended to convey some of the Defence Research Agency's initial ideas on extensions required in ANDF to support languages other than C. It was also passed to the ANDF technical discussions mailing list (andf-techosf.org) on 13th May 1993.

[41] Peter A. Darnell and Philip E. Margolis. *Software Engineering in C.* Springer Books on Professional Computing. Springer-Verlag, 1988.

[42] DEC Systems Research Centre. SRC Modula-3 system. Available as <ftp://gatekeeper.dec.com/pub/DEC/Modula-3/release>.

[43] S. Deering. Host extenstions for IP multicasting. Technical Report RFC 1112, IETF Network Working Group, August 1989. Available as <URL:ftp://ds.internic.net/rfc/rfc1112.txt>.

[44] S. Deering. MBONE: The multicast backbone. CERFNet Seminar, May 1993. Available as <URL:ftp://parcftp.xerox.com/pub/net-research/cerfnet-seminar-slides.ps.Z>.

[45] Department of Computer Science, Glasgow University. The glasgow haskell compiler (ghc). Available as <URL:ftp://ftp.dcs.glasgow.ac.uk/pub/haskell/glasgow/*>.

[46] *Data Encryption Standard.* Washington, DC, fips pub 46-2 edition, December 1993. Available as <http://csrc.nist.gov/fips/fips46-2.tx>.

[47] W. A. Doeringer, D. Dykeman, M. Kaiserwerth, B. W. Meister, H. Rudin, and R. Williamson. A survey of lightweight transport protocols for high speed networks. *IEEE Transactions on Communications*, 38(11):2025–2039, 1990.

[48] Andy Doherty. alt.comp.fsp Frequently Asked Questions, February 1994. Posted twice monthly to the alt.comp.fsp USENET newsgroup.

[49] G. Domann. A B-ISDN system concept and the BERKOM-testnetwork. In H.-J. Bullinger, E. N. Protonotarios, D. Bouwhuis, and F. Reim, editors, *EURINFO '88: First European Conference on Information Technology for Organisational Systems.* Addison-Wesley, May 1988.

[50] Fred Douglis and John Ousterhout. Transparent process migration: Design alternatives and the Sprite implementation. *Software: Practice and Experience*, 21(7), July 1991.

[51] *Digital Signature Standard (DSS).* Washington DC, USA, fips pub 186-1 edition, May 1994. Available as <URL:http://csrc.nist.gov/fips/fip186-1.txt>.

[52] Hans Eriksson. MBONE: The multicast backbone. *Communications of the ACM*, 37(8):54–60, August 1994.

[53] Joseph R. Falcone and Joel S. Emmer. A programmable interface language for heterogeneous distributed systems. Technical Report DEC-TR-371, Digital Equipment Corporation, August 1986.

[54] International Organization for Standardization and International Electrotechnical Committee. Information processing systems – open systems interconnection – transport protocol specification. international standard 8073, 1986. ITU-T Recommendation X.224 contains a *very* slightly altered description of the contents of this standard. X.224 is available as <gopher://info.itu.ch/11/.1/itudoc/public/gophertree/.1/.itu-t/.rec/.x/.24983>.

[55] Richard L. Ford. GANDF: A GCC-based ANDF translator. Technical report, Open Software Foundation, January 26th 1993.

[56] Robert B. Fougner. Public key standards and licenses. Technical Report RFC 1170, IETF Network Working Group, January 1991. Available as <ftp://ds.internic.net/rfc/rfc1170.txt>.

[57] Ron Frederick. nv: Network video tool, 1994. Available from <ftp://parcftp.xerox.com/pub/net-research/>.

[58] L. Gauffin, L. Hkansson, and B. Pehrson. Multi-gigabit networking based on PTM. *Computer Networks and ISDN Systems*, 24(2):119–130, April 1992.

[59] George Gilder. Into the fibresphere. *Forbes ASAP*, December 1993.

[60] Dave Gillespies. p2c Pascal to C translator. Available as <URL:ftp://csvax.cs.caltech.edu/pub/p2c-1.20.tar.z>.

[61] United States Government. The house-senate compromise version of S.272, the high performance computing act, 1991.

[62] Peter Grogono. *Programming in Pascal*. Computer Science. Addison-Wesley, 1984.

[63] P. Gunningberg, M. Bjorkman, E. Nordmark, S. Pink, and P. Sjodin. Application protocols and performance benchmarks. *IEEE Communication Magazine*, 27(6):30–36, June 1989.

[64] N. Haller. The S/KEY one-time password system. Technical Report RFC 1760, IETF Network Working Group, February 1995.

[65] Neil M. Haller. The S/Key one-time password system. In *Proceedings of the ISOC Symposium*. Bellcore, Morristown, New Jersey, 1993.

[66] Judith S. Hurwitz. OSF's ANDF: The key to shrinkwrapped software? *UNIX in the Office*, October 1991.

[67] Adobe Systems Inc. *PostScript Language Reference Manual*. Addison-Wesley, 1985.

[68] Sun Microsystems Inc. *UNIX Manual Section 8C: rexd*, SunOS 4.1.3 U1B edition, September 1987.

[69] Sun Microsystems Inc. XDR: External data representation standard. Technical Report RFC 1014, IETF Network Working Group, June 1987. Available as <URL:ftp://ds.internic.net/rfc/rfc1014.txt>.

[70] Sun Microsystems Inc. *UNIX Manual Section 1C: on*, SunOS 4.1.3 U1B edition, September 1988.

[71] Sun Microsystems Inc. NFS: Network file system protocol specification. Technical Report RFC 1094, IETF Network Working Group, March 1989. Available as <ftp://ftp.ds.internic.net/rfc/rfc1094.txt>.

[72] Sun Microsystems Inc. *UNIX Manual Section 2: socket*, SunOS 4.1.3 U1B edition, January 1990.

[73] Sun Microsystems Inc. *UNIX Manual Section 2V: pipe*, SunOS 4.1.3 U1B edition, January 1990.

[74] International Computer Science Institute. Sather programming language and environment. Available as <ftp://ftp.icsi.berkeley.edu/pub/sather/sa-0.2i.tar.Z>.

[75] International Standards Organisation and International Electrotechnical Committee. Information technology – programming languages – pascal, 1990. This version is a revision and redesignation of ANSI/IEEE 770X3.97-1983.

[76] ITU-T. Recommendation X.208 - specification of abstract syntax notiation one (ASN.1), 1993. Originally released in 1988 by the then CCITT. Available as <URL:gopher://info.itu.ch/11/.1/itudoc/public/gophertree/.1/.itu-t/.rec/.x/.24177>.

[77] ITU-T. Recommendation X.209 - specification of basic encoding rules for abstract syntax notation one (ASN.1), 1993. Originally released in 1988 by the then CCITT. Available as <URL:gopher://info.itu.ch/11/.1/itudoc/public/gophertree/.1/.itu-t/.rec/.x/.22887>.

[78] V. Jacobson, R. Braden, and D. Borman. TCP extensions for high performance. Technical Report RFC 1323, IETF Network Working Group, May 1992. Available as <URL:ftp://ds.internic.net/rfc/rfc1323.txt>.

[79] Van Jacobson. traceroute, December 1988. Available as <ftp://ftp.ee.lbl.gov/traceroute.tar.Z>.

[80] Van Jacobson. sd: The session directory tool v1.13beta, March 1993. Binaries for SPARC, DEC MIPS and Alpha, SGI MIPS and HP PA architectures available from <URL: ftp://ftp.ee.lbl.gov/conferencing/sd/>.

[81] Van Jacobson and Steve McCanne. vat: Visual audio tool, 1994. Available from <ftp://ftp.ee.lbl.gov/conferencing/vat/>.

[82] Bill Janssen, Denis Severson, and Mike Spreitzer. *ILU Reference Manual.* Xerox Corporation, 1.6.4 edition, May 1994. Available as <URL:ftp://parcftp.parc.xerox.com/pub/ilu/1.6.4/ilu-manual-1.6.4.ps.Z>.

[83] Sverre Johansen, Stein Krogdahl, and Terje Mjos. User's and installation guide, portable Simula system based on C. Technical report, Department of Informatics, University of Oslo, Norway, January 1991.

[84] Andrew Johnson, James Loveluck, and Ira Goldstein. The ANDF technology program at the OSF RI. Technical report, Open Software Foundation, December 8th 1992.

[85] S. C. Johnson. A portable compiler: Theory and practice. In *Proceedings of Fifth Annual ACM Symposium on Principles of Programming Languages*, January 1978.

[86] Mark Jones. Gofer. Available as <URL:ftp://nebula.cs.yale.edu/pub/haskell/gofer>.

[87] Mark G W Jones, Søren-Aksel Sørensen, and Steve R Wilbur. Protocol design for large group multicasting: the message distribution protocol. *Computer Communications*, 14(5):287–297, June 1991.

[88] Brewster Kahle and Harry Morris. Source description structures. Technical Report Unnumbered internal Thinking Machines technical report included in the WAIS-8-b5 public release, Thinking Machines Corporation, February 1991.

[89] B. Kaliski. The MD2 Message-Digest algorithm. Technical Report RFC 1319, IETF Network Working Group, April 1992. Available as <URL:ftp://ds.internic.net/rfc/rfc1319.txt.

[90] B. Kaliski. Privacy enhancement for internet electronic mail: Part iv: Key certification and related services. Technical Report RFC 1424, IETF Network Working Group, February 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc1424.txt.

[91] S. Kent. Privacy enhancement for internet electronic mail: Part ii: Certificate-based key management. Technical Report RFC 1422, IETF Network Working Group, February 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc1422.txt.

[92] Brian W. Kernighan and Dennis M. Ritchie. *The C Programming Language*. Prentice-Hall, 1978.

[93] L. Kleinrock. The latency/bandwidth tradeoff in gigabit networks. *IEEE Communications Magazine*, 30(4):36–40, April 1992.

[94] J. P. Knight and S. P. Guest. ANDF as an intermediate code for late-binding RPC. Technical Report TR-CS-853, Loughborough University of Technology, Ashby Road, Loughborough, Leics, UK, November 1993.

[95] J. Kohl and C. Neuman. The Kerberos network authentication service (v5). Technical Report RFC 1510, IETF Network Working Group, September 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc1510.t xt>.

[96] T. F. LaPorta and M. Schwartz. Architectures, features and implementation of high-speed transport protocols. *IEEE Network Magazine*, pages 14–22, May 1991.

[97] C. Lecht. *The Programmer's PL/1*. McGraw-Hill, 1968.

[98] Samual J. Leffler, Marshall Kirk McKusick, Michael J. Karels, and John S. Quarterman. *The Design and Implementation of the 4.3BSD UNIX Operating System*. Series in Computer Science. Addison-Wesley, 1989.

[99] J. Linn. Privacy enhancement for internet electronic mail: Part i: Message encryption and authentication procedures. Technical Report RFC 1421, IETF Network Working Group, February 1993. Available as <URL:ftp://ds.internic.net/rfc/rfc1421.txt.

[100] Andy Lowry. Hermes bytecode to C translator. Available as <URL:ftp://software.watson.ibm.com/pub/hermes/*>.

[101] Michael R. Macedonia and Donald P. Brutzman. Mbone provides audio and video over the Internet. *IEEE Computer*, pages 30–36, April 1994. Available as <URL:ftp://taurus.cs.nps.navy.mil/pub/mbmg/mbone.html>.

[102] Stavros Macrakis. From UNCOL to ANDF: Progress in standard intermediate languages. Technical report, Open Software Foundation, 1993.

[103] Stavros Macrakis. Porting to ANDF. Technical report, Open Software Foundation, 22nd January 1993.

[104] DoD DREN Program Management. Defense research and engineering network (DREN) executive summary, April 1994. Available as <http://www.arl.mil/HPCMP/DREN/drenexe3.html>.

[105] Evangelos P. Markatos and Thomas J. LeBlanc. Load balancing vs. locality managment in shared memory multiprocessors. Technical Report TR399, University of Rochester, Rochester, New York NY 14627 USA, October 1991.

[106] John McCarthy. Recursive functions of symbolic expressions and their computation by machine, part I. *Communications of the ACM*, 3(4), 1960.

[107] John McCarthy. History of LISP. *ACM SIGPLAN Notices*, 13(8), 1978. Also in *History of Programming Languages*, Richard L. Wexelblat (Ed.), Academic Pres, New York.

[108] John McCarthy, P. W. Abrahams, D. J. Edwards, T. P. Hart, and M. I. Levin. *LISP 1.5 Programmers Guide.* MIT Press, Cambridge, Massachusetts, USA, 1962.

[109] K. McCloghrie and M. T. Rose. Management information base for network management of TCP/IP-based internets. Technical Report RFC 1066, IETF Network Working Group, August 1988.

[110] Gennady Medvinsky and B. Clifford Neuman. Netcash: A design for practical electronic currency on the internet. In *Proceedings of the First ACM Conference on Computer and Communication Security.* Association for Computing Machinery, November 1993.

[111] Merit. Internet domain survey hosts list, August 1994. In July 1994, 3,212,000 hosts were found on the global Internet by Mark Lottor's ZONE program (RFC 1296). Available from <URL:ftp://nis.nsf.net/nsfnet/statistics/history.hosts>.

[112] R.M. Metcalfe and D.R. Boggs. Ethernet: Distributed packet switching for local computer networks. *Communications of the ACM*, 19(7):395–404, July 1976.

[113] Steve Miller and Clifford Neuman. *Kerberos UNIX manual pages*, 1986. Available as part of the Kerberos distribution from <URL://aeneas.mit.edu/pub/ATHENA/kerberos/>.

[114] mtc: A Modula-2 to C translator. Available as <URL:ftp://rusmv1.rus.uni-stuttgart.de/soft/unixtools/compilerbau/mtc.tar.Z>.

[115] Todd Montgomery. RMP - reliable multicast protocol - documents, 1995.

[116] National Centre for Supercomputer Applications. *The Common Gateway Interface*, 1993. Available as <URL:http://hoohoo.ncsa.uiuc.edu/cgi/overview.html>.

[117] A. N. Netravali, W. D. Roome, and K. Sabnani. Design and implementation of a high speed transport protocol. *IEEE Transactions on Communications*, 38(11):2010–2024, 1990.

[118] B. Clifford Neuman and Steven Seger Augart. The Prospero protocol. Technical Report Version 5, Revision 0.3b, USC Information Sciences Institute, 4676 Admiralty Way, Marina del Rey, California 90292–6695, U.S.A., February 1993. The latest revision of this document is available as <URL:ftp://prospero.isi.edu/pub/prospero/doc/prospero-protocol.PS.Z>.

[119] David A. Nichols. Using idle workstations in a shared computing environment. *ACM Operating Systems Review*, 21(5):5–12, 1987.

[120] High Level Protocol Group of the DCPU. *A Network Independent File Transfer Protocol*, February 1981.

[121] The JTP Working Party of the DCPU. *A Network Independent Job Transfer and Manipulation Protocol.* Department of Trade and Industry, London, UK, September 1981.

[122] SHARE Ad-Hoc Committee on Universal Languages. The problem of programming communications with changing machines. *Communications of the ACM*, 1(8 and 9), August and September 1958.

[123] International Standards Organisation and International Electrotechnical Committee. Information processing systems – open systems interconnection – file transfer, access and management – part 1: General introduction. Technical Report ISO 857-1:1988, ISO, April 1988.

[124] John K. Ousterhout. *Tcl and the Tk Toolkit.* Addison-Wesley, Reading, Massachusetts, USA, 1994.

[125] John K. Ousterhout. Winter USENIX presentation on TCL, January 1995.

[126] John K. Ousterhout, Andrew R. Cherenson, Frederick Douglis, Michael N. Nelson, and Brent B. Welch. The Sprite network operating system. *IEEE Computer*, February 1988.

[127] Frank G. Pagan. Converting interpreters into compilers. *Software Practise and Experience*, 18(6), June 1988.

[128] C. Partridge and G. Trewitt. HEMS variable definitions. Technical Report RFC 1024, IETF Network Working Group, October 1987.

[129] C. Partridge and G. Trewitt. High-level entity management protocol (HEMP). Technical Report RFC 1022, IETF Network Working Group, October 1987.

[130] C. Partridge and G. Trewitt. High-level entity management system (HEMS). Technical Report RFC 1021, IETF Network Working Group, October 1987.

[131] Craig Partridge. *Late-Binding RPC: A Paradigm for Distributed Computation in a Gigabit Environment.* PhD thesis, Computer Science, Harvard University, Cambridge, Massachusetts, USA, March 1992.

[132] Craig Partridge. *Gigabit Networking.* Addison-Wesley, Reading, Massachusetts, USA, 1994.

[133] N. E. Peeling. ANDF features and benefits. Technical report, Defence Research Agency, 1992.

[134] N. E. Peeling. TDF facts & figures. Technical report, Defence Research Agency, November 1992.

[135] N. E. Peeling. TDF specification. Technical report, Defence Research Agency, September 1992.

[136] B. Pehrson and S. Pink. MultiG: A swedish research programme on multimedia applications in gigabit networks. In *1st International Workshop on Operating System Support for Audio and Video*, Berkeley, CA, USA, November 1990. ICSI.

[137] J. Postel. Simple mail transfer protocol. Technical Report RFC 821, IETF Network Working Group, August 1982. Available as <URL:ftp://ds.internic.net/rfc/rfc821.txt.

[138] J. Postel and J. Reynolds. File transfer protocol. Technical Report RFC 959, IETF Network Working Group, October 1985. Available as <ftp://ds.internic.net/rfc/rfc959.txt>.

[139] Jon Postel. User datagram protocol. Technical Report RFC 768, IETF Network Working Group, August 1980. Available as <URL:ftp://ds.internic.net/rfc/rfc768.txt>.

[140] Jon Postel. Transmission control protocol. Technical Report RFC 793, IETF Network Working Group, September 1981. Available as <ftp://ftp.ds.internic.net/rfc/rfc793.txt>.

[141] J. Prevost. High bandwidth services and ISDN. In R. Speth, editor, *Procedings of EUTECO '88; Participants Edition*, pages 679–693, Amsterdam, Netherlands, 1988. North-Holland.

[142] Protocol Engines Inc., Santa Barbara, CA, USA. *XTP Protocol Definition*, revision 3.6 edition, January 1992.

[143] Tom Pyke. NOAA high performance computing and communications. In *ARPA High Performance Computing and Communications Symposium*. ARPA, March 1994. Available as <URL:http://hpcc1.hpcc.noaa.gov/hpcc/slides.html>.

[144] E. Rescorla and A. Schiffman. The secure hypertext transfer protocol. Technical report, Enterprise Integration Technologies, June 1994. Available as <URL:http://www.commerce.net/information/standards/drafts/shttp.txt>.

[145] R. Rivest. The MD4 Message-Digest algorithm. Technical Report RFC 1320, IETF Network Working Group, April 1992. Available as <URL:ftp://ds.internic.net/rfc/rfc1320.txt.

[146] R. Rivest. The MD5 Message-Digest algorithm. Technical Report RFC 1321, IETF Network Working Group, April 1992. Available as <URL:ftp://ds.internic.net/rfc/rfc1321.txt.

[147] Marshall T. Rose. *The Open Book: A Practical Perspective on OSI.* Prentice-Hall, 1989.

[148] Marshall T. Rose and Nathaniel Borenstein. A model for enabled mail (em), July 1994. Working draft. Available as part of <URL:ftp.ics.uci.edu/mrose/safe-tcl/safe-tcl.tar.Z>.

[149] R. R. Rowlingson and N. E. Peeling. Commonly asked questions about ANDF. Technical report, Defence Research Agency, 1992.

[150] Peter Van Roy. 1983-1993: The wonder years of sequential prolog implementation. *Journal of Logic Programming*, 1994. Also available as a DEC Paris Research Laboratory technical report available from <ftp://duck.dfki.uni-sb.de/pub/ccl/dfki-saarbruecken/SequentialPrologImp.ps.Z>.

[151] Harry I. Rubin. *Process Placement for Distributed Computations.* PhD thesis, Computer Science, University of California at Berkeley, CA, USA, 1992.

[152] Jean E. Sammet. *Programming Languages: History and Fundamentals.* Prentice-Hall, 1969.

[153] M. F. Schwartz and P. G. Tsirigotis. Experience with a semantically cognizant internet white pages directory tool. *Journal of Internetworking Research and Experience*, pages 23–50, March 1991.

[154] Stuart Sechrest. A client-server shell architecture for distributed programming. Technical Report UCB//CSD-88-457, University of California at Berkeley Computer Science Department, 1988. Available as <ftp://tr-ftp.cs.berkeley.edu/pub/tech-reports/csd/csd-88-457/>.

[155] Behrooz Shirazi and A R Hurson. Guest editors' introduction. *Journal of Parallel and Distributed Computing*, 16(4):271–275, 1992.

[156] *Secure Hash Standard.* Washington DC, USA, fips pub 180-1 edition, May 1994. Available as <URL:http://csrc.nist.gov/fips/fip180-1.txt>.

[157] Silicon Graphics Inc. *IRIX Network Programming Guide*, 11/93 edition, 1991.

[158] M. Skov. Implementation of physical and media access protocols for high speed networks. *IEEE Communications Magazine*, 27(6):45–53, June 1989.

[159] Brian C. Smith and Lawrence A. Rowe. TCL-DP v3.2, August 1994. Distribution available from <URL:ftp://harbor.ecn.purdue.edu/pub/tcl/extensions/tcl-dp3.2.tar.gz>.

[160] Jerry D. Smith. *An introduction to Scheme.* Prentice-Hall, London, 1988.

[161] Simon E. Spero. Analysis of HTTP/1.0 performance problems, July 1994. Posted to the mailing list www-talkinfo.cern.ch with message ID <9407160847.AA03555tipper.oit.unc.edu>.

[162] Richard M. Stallman. *Using GNU CC Version 2.3.1.* GNU, 1992.

[163] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Transactions on Programming Languages and Systems*, 12(4):537–565, October 1990.

[164] James W. Stamos and David K. Gifford. Remote evaluation. *ACM Trans. Programming Languages and Systems*, 12(4):537–565, October 1990.

[165] Guy L. Steele. *Common Lisp: The Language, 2nd Edition.* Digital Press, 1990.

[166] R.G. Stone and D.J. Cooke. *Program Construction.* Cambridge University Press, 1987.

[167] Bjarne Stroustrup. *The C++ Programming Language.* Addison-Wesley series in computer science. Addison-Wesley, 1986.

[168] Inc. Sun Microsystems. *UNIX Manual Section 1: hostid*, SunOS 4.1.3 U1B edition, September 1987.

[169] Inc. Sun Microsystems. *UNIX Manual Section 8C: ping*, SunOS 4.1.3 U1B edition, May 1988.

[170] Tanel Tammet. Hobbit Scheme to C translator. Available as <URL:ftp://altdorf.ai.mit.edu/archive/scm/hobbit1.tar.Z>.

[171] A. S. Tanenbaum and R. van Renesse. A critique of the remote procedure call paradigm. In *EUTECO '88 Proceedings, Particpants Edition*, pages 775–783, Amsterdam, Netherlands, 1988. North-Holland.

[172] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van. Rossum. Experiences with the amoeba distributed operating system. *Communications of the ACM*, 33(12):46–64, December 1990.

[173] Andrew S. Tanenbaum, Robbert van Renesse, Hans van Staveren, Gregory J. Sharp, Sape J. Mullender, Jack Jansen, and Guido van Rossum. Experiences with the Amoeba distributed operating system. *Communications of the ACM*, 33(12):46–64, December 1990.

[174] Andrew S. Tanenbaum, Hans van Staveren, E. G. Keizer, and Johan W. Stevenson. A practical tool kit for making portable compilers. *Communications of the ACM*, 26(9):654–660, September 1983.

[175] David Tarditi, Anurag Acharya, and Peter Lee. No assembly required: Compiling Standard ML to C. Technical Report CMU-CS-90-187, School of Computer Science, Carnegie Mellon University, Pittsburgh, PA, 15213, USA., November 1990.

[176] B. H. Tay and A. L. Ananda. A survey of Remote Procedure Calls. *ACM Operating Systems Review*, 24(3):68–79, July 1990.

[177] Joe Touch. Report on MD5 performance. Technical Report draft-touch-md5-performance-00, ISI, December 1994. Work in progress. Available as <URL:ftp://ds.internic.net/internet-drafts/draft-touch-md5-performance-00.txt>.

[178] G. Trewitt and C. Partridge. HEMS monitoring and control language. Technical Report RFC 1076, IETF Network Working Group, November 1988.

[179] International Telecommunications Union. Information technology – open systems interconnection – basic reference model: The basic model. Technical Report Recommendation X.200, ITU-T,

July 1994. Available as <gopher://info.itu.ch/11/.1/itudoc/public/gophertree/.1/.itu-t/.rec/.x/.25094/.14704.zip>. Previously published by the International Standards Organisation as ISO/IEC IS 7498-1.

[180] R. Unran, M. Stumm, and O. Krieger. Hierarchical clustering: A structure for scalable multiprocessor operating system design. Technical Report CSRI-268, University of Toronto, March 1992.

[181] Ron Unrau, Michael Stumm, and Orran Krieger. Hierarchical clustering: A structure for scalable multiprocessor operating system design. Technical Report CSRI-268, Computer Systems Research Institute, University of Toronto, Toronto, Canada, M5S 1A4, March 1992.

[182] Guido van Rossum. *Python Reference Manual.* Dept. CST, CWI, 1090 GB Amsterdam, Netherlands, 1.0.3 edition, July 1994. Available as <URL:http://www.cwi.nl/ guido/python-ref/ref.html>.

[183] Various authors at Bellcore. f2c. Available as <URL:ftp://research.att.com/dist/f2c>.

[184] Zvonko G. Vranesic, Micheal Stumm, David M. Lewis, and Ron White. Hector: A hierarchically structured shared-memory multiprocessor. *IEEE Computer*, 24(1):72–79, January 1991.

[185] Larry Wall and Randal L. Schwartz. *Programming Perl.* Nutshell Handbook. O'Reilly and Associates, 1991.

[186] Brent B. Welch. The Sprite remote procedure call system. Technical Report UCB/CSD 86/302, University of California at Berkeley, June 1986.

[187] Brent Ballinger Welch. *Naming, State Management, and User-Level Extensions in the Sprite Distributed File System.* PhD thesis, Computer Science Division (EECS), April 1990. Also available as Technival Report UCB/CSD 90/567.

[188] Brian Whetten, Todd Montgomery, and Simon Kaplan. A high performance totally ordered multicast protocol. In *Proceedings of the Dagstuhl Dis-*

tributed Systems Workshop. Springer Verlag, September 1994. Available as <ftp://research.ivv.nasa.gov/pub/doc/RMP/RMP_Dagstuhl.ps>.

[189] A. Whitcroft, N. Williams, and P. E. Osmon. The wide area data space. Technical Report TCU/SARC/1993/6, CS Dept, City University, London, UK, 1993. Available from <URL:ftp://ftp.cs.city.ac.uk/papers/93/sarc93-6.ps.Z>.

[190] James E. White. A high-level framework for network-based resource sharing. In *National Computer Conference Proceedings*, volume 45, Montvale, NJ 07645, USA, 1976. AFIPS.

[191] ANSI Committee X3J11. *ANSI Standard X3.159-1989.* American National Standards Institute, 1990.

[192] Jian Xu and Ka Hwang. Heuristic methods for dynamic load balancing in a message-passing multicomputer. *Journal of Parallel and Distributed Computing*, 18(1):1–13, 1993.

[193] Edward R. Zayas. Attacking the process migration bottleneck. *ACM Operating Systems Review*, 21(5):13–24, 1987.

[194] Yahui Zhu. Efficient processor allocation strategies for mesh-connected parallel computer. *Journal of Parallel and Distributed Computing*, 16(4):328–337, 1992.