Towards the rapid network-wide deployment of new application specific network protocols, using application level active networking.

Atanu Ghosh

A dissertation submitted in partial fulfillment

of the requirements for the degree of

Doctor of Philosophy

of the

University of London.

Department of Computer Science

University College London

January 8, 2002

In memory of my father Shyama Prosad Ghosh.

“Sometimes a scream is better than a thesis.”

- Ralph Waldo Emerson (1803-1882)

Acknowledgements

I am indebted to my supervisors Jon Crowcroft and Michael Fry. Jon for his patience, incredible enthusiasm and encouragement. Michael for the work that we did together. Mark Handley for allowing me more time than either of us had expected. Nermeen Ismail for her support. My mother, Juthika Ghosh, for everything.

Abstract

In this thesis I try to show that traditional transport protocols (such as TCP) poorly match the requirements of today's applications.

Firstly, I develop two new user level protocols using Application Layer Framing (ALF) [20] concepts in order to test this hypothesis. A simple remote login program and a protocol to deliver mailing lists using multicast. In both cases I am able to show that application level protocols utilise less network resources than their traditional counterparts as well as providing improved responsiveness to the user.

The development and deployment of new protocols is both difficult and time consuming. I consider a new mechanism for the deployment of protocols. Rather than attempting to deploy them in the end systems, new protocols entities are dynamically deployed into the network. These protocol entities are called *proxylets* and are written in the programming language Java. A whole infrastructure, *funnelWeb*, [34], has been built to allow the deployment of *proxylets*. A number of diverse applications are then built around *proxylets* to show benefits which can be derived from such a scheme. In order to make the *funnelWeb* infrastructure useful, when globally deployed, a routing infrastructure is designed and partially built.

Contents

1 Introduction **19**

1.1 Chapter outline 22

1.2 Discussion 23

1.3 Experimental Results 25

2 Simple login (*slogin*) **29**

2.1 Introduction 29

2.2 *Slogin* Protocol Overview 32

2.3 *Slogin* Protocol Requirements 33

2.4 The *Slogin* Protocol 34

2.4.1 Packet Format 34

2.4.2 Data 36

2.4.3 The *Slogin* Protocol 37

2.5 Implementation Issues 38

2.5.1 Control packets 38

2.5.2 Retransmission 40

2.5.3 Bind and connect problems with UDP 40

2.5.4 User Privileges 42

2.5.5 Testing 43

2.5.6 Security 43

2.6 Experiments 46

2.7 Results 47

2.8 Enhancements 47

2.9 Discussion 48

2.10 Conclusions 49

3 MMD: Multicast Mail Delivery 51

3.1 Introduction 51

3.2 Design Basics 54

3.3 The Structure 55

3.4 The Protocol 56

3.4.1 Status message 57

3.4.2 NACK message 57

3.4.3 Data message 58

3.4.4 Retransmission Scheme 59

3.5 Related Work 63

3.6 Discussion 63

3.7 Conclusions 64

4 JavaRadio: an application level active network 67

4.1 Introduction 67

4.2 Proposal for Mobile Protocol Stacks 68

4.2.1 Application Choice 69

4.2.2 Significance 70

4.3 Experiment Description 71

4.4 Component Overview 73

4.4.1 YAAT - Yet Another Audio Tool 73

4.4.2 Proxy Server 75

4.4.3 Control Interface 76

4.4.4 Protocol Server 76

4.5 Implementation Issues 77

4.5.1 Portability 77

4.5.2 Stability 77

4.5.3 Threads 78

4.5.4 Header Processing 78

4.6 Experimental Outcomes 81

4.6.1 Local Area Experiment 82

4.6.2 Wide Area Experiment 82

4.6.3 Multicast Reception 84

4.7 Related Work 84

4.8 Discussions and Conclusions 85

5 Application Level Active Networking 89

5.1 Introduction 89

5.2 ALAN Overview 92

5.2.1 Basic ALAN 92

5.2.2 ALAN Enhancements 95

5.3 *Proxylets* 97

5.3.1 Design 97

5.3.2 Implementation 98

5.3.3 Lessons learned from deployment 100

5.3.4 Future directions 101

5.4 Execution Environment for *Proxylets* -

EEPs 102

5.4.1 Design 102

5.4.2 Implementation Issues 104

5.4.3 Lessons learned from deployment 105

5.5 Controlling *Proxylets* -

The graphical user interfaces 106

5.6 *System Proxylets* 107

5.6.1 Routing 108

5.6.2 Error Handling 109

5.7 Performance Measurements 110

5.7.1 EEP Location 110

5.7.2 *Proxylet* Load Time 110

5.7.3 Execution Time 111

5.8 Charging 113

5.9 Related Work 114

5.10 Deployment 115

5.11 *Proxylets* 116

5.12 Conclusions 117

6 An Architecture for Application Layer Routing 119

6.1 Introduction 119

6.2 Application Layer Active Networking 121

6.2.1 WWW cache *proxylet* 121

6.2.2 TCPbridge 122

6.2.3 VOIP gateway 123

6.2.4 Multicast 124

6.3 Application Layer Routing Architecture 124

6.4 Discovery 127

6.4.1 Discovery phase 127

6.5 Routing exchanges 132

6.5.1 Connectivity mesh 133

6.5.2 Snooping for network state 134

6.5.3 Self Organising Application-level Routing - SOAR 134

6.6 Related Work 138

7 Conclusion 141

7.1 Future Work 146

A Glossary 149

B *Flakeway* 155

B.1 Implementation 158

B.2 Conclusions 161

C TCP Evolution 163

C.1 Application modifications 163

C.1.1 Netscape - multiple streams 164

C.1.2 HTTP - single connection 164

C.2 Implementation modifications 165

C.2.1 Transaction TCP T/TCP 165

C.2.2 Large Windows 165

C.2.3 Selective Acknowledgement - SACK 165

C.2.4 Congestion Control 166

C.2.5 Unreliable Delivery 166

D RC4 Implementation 169

List of Figures

2.1 *slogin* packet format 34

3.1 The Structure of MMD 55

4.1 Framework 73

4.2 Yet Another Audio Tool 74

4.3 Control Interface 76

4.4 RTP header 79

4.5 Header Processing in C 79

4.6 Header Processing in Java 80

5.1 Text Compression 93

5.2 Interface implemented by a *proxylet* 99

6.1 Text Compression 122

B.1 Possible actions 156

B.2 Example Packet count table 157

B.3 Example Time sequence table 157

B.4 Ping Trace. UCL - INRIA 158

B.5 Flakeway workstation 159

B.6 Network configuration 160

D.1 rc4.h 169

D.2 rc4.c 170

List of Tables

2.1 *slogin* state transitions 39

5.1 Compression Times 112

6.1 Terms in the Padhye TCP Equation 136

Chapter 1

Introduction

We are stifled in the process of developing new more efficient network protocols due to the increasing difficulty in deployment.

In the statement above and in the title “network-wide protocols”, the protocols are not end-to-end transport or network protocols in the traditional sense. They are protocols which are specific to an application built, for example, on top of a connectionless transport protocol such as the User Datagram Protocol (UDP) [70]. These are protocols which are implemented by applications as opposed to protocols such as Transmission Control Protocol (TCP) [72] which are traditionally provided by the host operating system.

We argue that many networking applications are ill suited to using traditional transport protocols such as TCP. In Appendix C is a representative list of either changes that have been made to TCP or workarounds when using TCP. We suggest that it is more efficient to implement protocols that map directly onto an application’s requirements.

If high performance in terms of raw throughput is not a requirement, then a mechanism for speeding up the testing and deployment of an application specific protocol is to build it into the application. Thus the protocol is implemented in “user space” rather than the traditional approach of building it into the operating system. This is called Application Layer Framing (ALF) [20]. It is typically faster to send out a new application than to ship a new operating system.

If the hypothesis that building application specific protocols has value, then the next problem is: How are these application specific protocols developed and deployed? New protocols are often shunned for a number of reasons. The main difficulty is time: the time to implement, the time to test, the time to deploy. It is rare for software to be shipped without anomalies or bugs. Network protocols often go through a "bake-off" [73] process where different implementors can test their implementations against each other. Specifications especially in protocol areas can be difficult to follow and implement "correctly". Therfore, there is also the time to redeploy.

Hence ease of deployment of new protocols is a key component of their being adopted. My work focuses on this issue of deployment. By building two protocols using UDP as the transport no operating system modifications should be required. Although some bugs in implementations of UDP were found, protocols built entirely in user space are generally faster to deploy than protocols requiring operating system implementations and deployments.

To help simplify the creation of user space protocols an initial examination was made to specify building blocks which could be used [32]. As part of this examination two application specific protocols were built using Application Layer Framing (ALF) [20] concepts.

Rather than focus on the implementation of the protocols it became apparent that deployment was the key. What if only a single implementation could be used by all applications which required a particular protocol then the goal of rapid protocol deployment could be met.

Using Java [9], a relatively new programming language at the time, some experiments were performed to create protocol stacks which could then be dynamically downloaded, on demand, into end systems by applications.

These experiments were successful in terms of performance, however, specifying a general API from application to protocol stack was not a simple process. The separation of the application from the protocol stack proved to be difficult.

Instead of attempting to download code into an end system to aid in the deployment of a protocol a whole protocol entity was downloaded into the network. In the experiments that were performed the end system stacks were not modified in any significant way. This led to the development of Application Level Active Networking (ALAN).

Thus the rapid deployment of protocols was achieved by loading dynamic code into the “network”. This is not loading code into routers as advocated by the active network community [83]. In the active networking model this code can then manipulate packets travelling through these routers and these manipulations can be considered to be transparent to the end systems.

In the case of ALAN programs named *proxylets* are loaded into nodes on a network. These nodes are not routers. In an optimal situation they would be placed in the core of the network. Normally the *proxylets* would be explicitly placed into the path of a flow. This too is at variance with the usual active networking ideas, in which packets can be transparently manipulated.

1.1 Chapter outline

The initial work in this thesis, Chapters 2 and 3 were done as part of larger project, the Hipparch [5] project. Other project members were considering another aspect of the problem, automated protocol composition [23]. My initial work consisted of hand building integral applications and protocols. This was done to both verify that there was a benefit from this approach as well as trying to determine what could be learnt and extracted to form building blocks of future ALF applications. We consider two classic network applications. Chapter 2 describes an ALF based remote login protocol called *slogin*. As an enhancement over the applications telnet and rlogin available at the time, *slogin* also supported encryption. In Chapter 3 MMD describes a Multicast Mail Delivery protocol which was developed for the delivery of mailing lists. Both pieces of work develop protocols which are tightly matched to the applications requirements. These protocols are more efficient in terms of packet exchange than their TCP equivalents. Unfortunately there is the added complexity of having to design, verify and test these protocols

The JavaRadio Chapter, Chapter 4, describes a mechanism in which the protocol stack itself can be dynamically loaded by an application. It was also an experiment to discover if the then relatively new Java programming language could be used for real time processing. The work in this Chapter was presented at the Hipparch Workshop in 1997 in Uppsala, Sweden [35].

Chapter 5 on Application Level Active Networking describes an infrastructure called *funnelWeb* [34], which was designed and built to test some of the ideas presented. The work in this Chapter appeared in the Journal of Computer Networks [36].

The penultimate Chapter, Chapter 6, describes a natural extension to Application

Layer Active Networking - Application Layer Routing. This work was presented at the the International Workshop on Active Networking in 2000 in Tokyo, Japan.

The last Chapter, Chapter 7, is the conclusion.

1.2 Discussion

The two main applications are: *slogin* (a simple login program) and MMD a multicast mail delivery program, which used a reliable multicast protocol to deliver mail to email list. An infrastructure *flakeway* (Appendix B) was also built to test the *slogin* program.

We were able to show with the *slogin* and MMD programs that there is indeed a benefit to be derived from fitting an applications requirements directly to its own associated protocol.

It may be considered that this approach is limited as there is a lot of extra effort involved in building a protocol for each application as opposed to just using TCP or equivalent. In order to aid in the building of such applications we suggested some possible building blocks [32].

At this point rather than build a toolkit to construct ALF style applications, I became more interested in the proposition that very few people would ever need to design an ALF based protocol. Consider an application such a ftp. How many people have actually written an ftp client or daemon? Consider the most widely used protocol on the Internet, currently HTTP [25]. The W3C consortium actually provide a library to implement this protocol.

The next step was to consider building protocol libraries which could be used by applications. The challenges were to develop a generic API for an arbitrary set of protocol stacks and deal with version compatibility. For example, specifying a generic protocol API is a tricky process if out of order delivery is required.

The traditional approach to a protocol library may have been to design an API and implement a number of protocols. In the protocol library idea I wanted to address the ubiquitous version mismatch problem with protocol stacks.

Hence, the idea of a dynamic protocol stack was conceived. In order to address the protocol mismatch problem, why not dynamically load a protocol stack into an application at runtime? This solves many problems; only one implementation of a protocol, perhaps available from the appropriate standards body. There is an issue: how is the protocol stack made processor independent?

At this time a new programming language, Java, was becoming popular. A possible solution seemed to be use Java as it could be compiled into processor neutral byte codes. There was some concern at the time that since Java was an interpreted and garbage collected language it could not meet the performance requirements of a large set of applications.

However, Java had a lot to offer in terms of code portability. The holy grail of a single portable implementation of a protocol with no version incompatibility issues was too tantalising to be ignored. Though there was still the issue of performance.

In order to investigate the performance characteristics of using Java with a realistic application, an audio playout tool was written (Chapter 4). In order to test the RTP compliant playout tool an audio encoder was also written. It turned out that for a real time application such as audio playout Java met the performance criteria.

The next step should have been to define an API for protocol stacks. However, it became apparent that it would be more beneficial to start placing the code in the network, hence ALAN (Application Level Active Networking). A number of papers have been written is this area [35], [31], [36], and [33].

Having built an ALAN infrastructure it was possible to see that there were benefits in such a scheme. However, a significant limitation in the system is that it is a requirement to know beforehand the location of EEP's (Execution Environment for Proxylets, the nodes on which the active code executes). Having to know the location of EEP's before running code is not a solution which scales. Therefore some initial work was done in the area of Application Layer Routing (ALR) [33]. A two stage scheme was designed and implemented that addressed the issues of discovery and routing.

The original idea was to build tailored application specific protocols. In order to pursue this goal the final outcome was a mechanism to place code into the network to aid the performance and deployment of applications.

1.3 Experimental Results

One major problem with the work presented is being able to demonstrate that ALF and ALAN provide any benefit over traditional networking paradigms. One way of demonstrating benefit is to measure an existing system and then compare against the new proposed system.

In the *slogin* work (chapter 2) it is possible to reason that less resources are used for an *slogin* session over an equivalent rlogin session. In order to test the *slogin* protocol and to compare it against the rlogin protocol a testing harness was built. The testing harness, *flakeway,* is described in Appendix B. *Flakeway* was introduced into the path of a connection and its purpose was to delay or drop packets. Reordering could also be achieved; reordering is a subset of delay. A novel feature of *flakeway* was that it does not use a random drop probability. Rather, is it table driven. Each arriving packet is manipulated (in terms of delay or being dropped) by being compared against a table which determines how a packet is modified. This gives deterministic behaviour across test runs. If the input packets are identical across test runs the results are identical. The tables used to drive *flakeway* were derived from actual packet traces. The reasoning is that the only time any major variance will be seen in the performance of *slogin* against rlogin, will be when there is some level of loss and some transmission delay. There will be no perceptible difference in performance on a LAN with no loss or delay. Therefore in order to test *slogin* against traditional applications such as rlogin a wide area test needs to be performed. The conditions in the global Internet are continually changing. It would not really prove anything if we were to say that at a particular time *slogin* performed “better” than rlogin in a wide area test. In order to be able to generate repeatable tests *flakeway* was built. The tables for driving *flakeway* were derived by running pings to sites in remote countries. The delay, reordering and losses were then used to drive *flakeway*. Various factors such as variable packet sizes and extra load were not taken into account. *Flakeway* turned out to be an extremely useful tool for testing the *slogin* protocol as it was being developed. For example, the connection setup phase could be fairly exhaustively tested be delaying or dropping the initial packets on a connection. Unfortunately, it is not possible to justify that *slogin* performed better than rlogin by just performing test runs through *flakeway*.

A similar problem exists with the MMD work (chapter 3). It is difficult to show experimentally that this scheme is superior to the standard distribution schemes.

In both the *slogin* and the MMD case it is shown that protocols can be designed and deployed using ALF principles that are better mapped to the applications requirements.

The subsequent work in Application Level Application Networking (ALAN) and

Application Layer Routing (ALR) (Chapters 5 and 6) also exhibit this property that it is difficult to prove they are an improvement over current networking technologies.

I have not been able to prove that application specific protocols are the best or only way to deploy new applications. I have been able to demonstrate that there are benefits from building application specific protocols. I have also been able to demonstrate that it is possible to deploy new protocols by placing them dynamically into the network rather than in the end systems.

Chapter 2

Simple login (*slogin*)

2.1 Introduction

TCP is a general purpose communication protocol that attempts to satisfy the requirements of a diverse set of applications. It is a stream based protocol that transfers streams of octets. The reliable transmission of data provided by TCP simplifies application writing, which is why TCP is traditionally used by most Internet applications requiring reliable delivery. An example of such application is remote login where the two popular programs in use on the Internet: telnet [74] and rlogin [45] are both TCP based. This is despite the fact that a stream based protocol does not usually map easily onto the packet based nature of an application such as remote login. This usually results in failing to provide the correct trade off between delay and throughput that meets the application needs.

In a remote login application, one end of the connection transmits text typed by the user while the other end of the connection generates output based on the typed text as well as possibly echoing back what it has received.

Human users can read and generate data, through typing, at very low rates that are definitely lower than that supported by most networks today. Hence low latency rather than a high data throughput is the main requirement of a remote login application. However, there are occasions when a word processing application may need to send several kilobytes of data to update the display.

In an attempt to reduce the number of packets exchanged and hence keep the load on the network to a minimum, TCP implementations use two mechanisms: delayed ACK and the Nagle algorithm [12, 63] [1]. The goal is to reduce the number of packets exchanged by increasing the amount of information carried in each packet. Effectively this is achieved by delaying the transmission of packets to allow more information to be carried in each packet.

For a long delay path, using TCP decreases the responsiveness of the remote login connection. This is a well known side affect of the delayed ACK and the Nagle algorithm. The delayed ACK typically does not increase the latency. However, in situations where no data needs to be sent back, the delayed ACK will cause latency.

On the other hand, the Nagle algorithm typically means that at least two RTT times will pass before the echo of a command is received. The Nagle algorithm is simple and elegant. On packet transmission if there is no data in the transmit queue then the data is sent immediately. However, if there is data in the transmit queue then the new data is queued until the arrival of the ACK for the outstanding data or the retransmission timer expires. So a delay is imposed on the transmit queue to allow time for more data to build up in the transmit queue. When the ACK arrives or the retransmit timer expires, all queued data can be sent in one packet.

One of the examples from the Nagle request for comments (RFC [63]) is a link

[1] It should be noted that there are instances where the Nagle algorithm can interact extremely badly with applications [59]

with a round trip time of 5 seconds. The user generates 24 characters with an inter character gap of 200ms. The first character will be sent immediately while the other 23 characters will be queued awaiting the ACK for the first character. Typing 23 characters will take 4.6 seconds which is less than the 5 second round trip time. When the ACK for the first character arrives all 23 characters will be sent in the same packet. The number of packets sent has been reduced to a minimum of 2 packets, considering the no loss situation where no packets have been lost, as opposed to 24 data packets. The time to get the ACK in the Nagle case will be at least 10 seconds. If each character is sent in its own packet the time to get an ACK will be 24*.200 + 5 = 9.8 seconds. So in this example using the Nagle algorithm saves the sending of 22 data packets and only costs an extra 200ms. For the user in the Nagle case it will take 5 seconds for the first character to be echoed and 10 seconds after the first character being typed (or 9.8 seconds after the second character being typed) for the rest of the characters to be echoed. In the non Nagle case, assuming no loss, the echo will appear after one RTT delay, which is much more acceptable to the user. However, sending only one data byte per packet is extremely inefficient in terms of link usage. With a 20 byte IP header and a 20 byte TCP header, a 41 byte packet containing only one byte of data is inefficient in terms of link utilisation. Under heavy network congestion, many small data packets will increase the congestion.

Slogin is a program that has its communications stack built with consideration of the requirements of the application. *Slogin* was built to show that a hand crafted protocol could be made more responsive than TCP and still does not cost too much more than TCP in terms of link utilisation. Encryption is a value added feature of *slogin*.

The building of *slogin* also facilitated the identification of the tools needed for building specially tailored protocols for specific applications. One of the intended outcomes from this work was to ultimately generate a set of building blocks. These building blocks could then be used to build the next generation of application specific protocols. This direction was ultimately not pursued. Although a paper attempting to define useful building blocks was written in 1996: “Some Lessons Learned from Various ALF and ILP Applications” [32].

2.2 *Slogin* Protocol Overview

The *slogin* program is relatively simple. It uses its own packet format on top of UDP. The packet format contains header fields that allow the building of a reliable protocol. Various ideas from TCP implementations are used such as delayed ACKs and the mechanism for calculating the RTT. The *slogin* program generates more packets than an equivalent remote login program using TCP because the Nagle algorithm is not used. The Nagle algorithm in TCP is not just for remote login applications but also for any application that generates a few bytes at a time that can be amortised into larger packets. The Nagle algorithm is inappropriate for remote login sessions over long delay paths. In order to keep the response time to a minimum, the *slogin* protocol sends packets the moment characters are generated by the user. This increases the number of packets generated, however, in defence of this strategy, a single user typing a character every 200ms can only generate 5 characters per second resulting in 5 packets per second. An important aspect of the *slogin* protocol is that each packet carries all of the data that has not yet been acknowledged. Hence any packet loss will not necessarily result in an extra RTT of delay. This is another boost to the responsiveness of the protocol. A

TCP implementation can not make the decision to carry all data that has not yet been acknowledged in its packets because as a general purpose protocol it does not know the shape of the traffic generated by the application. On the other hand a specially tailored protocol can take full advantage of such knowledge. While the TCP protocol requires 20 bytes, assuming no options, the *slogin* protocol uses 7 bytes (currently more are used for ease of implementation) plus 8 bytes of UDP header to give a total of 15 bytes; a saving of 5 bytes per packet. The comparative header sizes are not given to imply that *slogin* is better then TCP because it uses fewer bytes. It just illustrates that an equivalent number of bytes is used.

2.3 *Slogin* Protocol Requirements

The goal of *slogin* is to develop a UDP-based transport protocol that is optimised for remote login applications over long delay links. A second requirement is the use of encryption for security purposes. A number of goals are identified for the protocol.

1. Most importantly, from the user point of view, the protocol needs to be more responsive than the traditional programs such as rlogin or telnet.

2. It is also important that the number of bytes exchanged using *slogin* is not significantly more than the number of bytes exchanged when using TCP based programs.

3. It is also a goal to achieve better interrupt and expediated data processing. The rlogin program uses TCP's urgent pointer (URG) to perform expedited data processing. One significant drawback of this approach is that the URG pointer is exactly that: a pointer into the stream. A particular point in the byte stream is

```
 0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Type of Packet                      |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Acknowledgement Number              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Length                              |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Sequence Number                     |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|                           Data                                |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 2.1: *slogin* packet format

marked as being urgent. Depending on the TCP implementation the urgent byte is not necessarily delivered before any other data.

2.4 The *Slogin* Protocol

2.4.1 Packet Format

The packet format is rather simple. It is made up of four fields followed by an optional payload (Figure 2.1).

Figure 2.1 shows the *slogin* packet format. The packet header is much larger than it needs to be. Each field is kept as a four byte quantity for ease of implementation. The type field can be reduced to three bits as there are only six packet types. The length field can also be removed as UDP gives the length of the packet. So all in all the header can be reduced by slightly more than 5 bytes. Each packet contains the same fixed size header for simple parsing independent of the packet type. Though controversial, there is no version field in the packet format. *Slogin* utilizes the simple expedient of using a new UDP port for major version changes, which is simpler than the use and check of a version number field. Such decision forces any *slogin* version to always use a fixed UDP port. However, there is no requirement to check version fields nor support backward compatibility. There is no interoperability concern.

A number of fields that are usually present in transport protocols are missing in *slogin*. There is no ID field to associate packets with sessions. Instead, the host and port fields contained in the UDP header are used for that purpose.

Slogin packets have no additional checksum apart from the checksum provided by UDP. However, a checkbyte is placed at the end of the payload to aid with the authentication of packets. As both TCP and UDP use the same checksum algorithm, both *slogin* and rlogin packets have the same protection against data corruption. The additional checkbyte in the *slogin* packets provides extra protection despite its main purpose being authentication rather than protection.

The packet format has the following fields:

Packet types

There are 7 packet types in *slogin*:

1. CR - Connection Request - Value 0

   A connection request packet.

2. CC - Connection Confirmation - Value 1

   A connection confirmation packet.

3. DATA - Data Packet - Value 2

   A data packet.

4. CONTROL - Control Packet - Value 3

   A control packet.

   This packet type conveys out of band information such as interrupt characters, terminal size , flow control information and terminal type.

   The current implementation does not support this packet type.

5. DR - Disconnect Request - Value 4

   A disconnect request packet.

6. DC - Disconnect Confirm - Value 5

   A disconnect confirmation packet.

7. RESET - Reset - Value 6

   A reset packet.

Acknowledgement Number

This field acknowledges the last byte received from the peer.

Length

This field specifies the length of the packet. It is a redundant field as the UDP API provides the same information.

Sequence Number

This is the sequence number of the first byte of data contained in the packet.

2.4.2 Data

The payload is encrypted by the RC4 stream cipher. There is also an additional check-byte trailing the data to facilitate packet authentication.

2.4.3 The *Slogin* Protocol

An *slogin* session can be in one of four states: IDLE, CONNECTING, CONNECTED and DISCONNECTING.

The simplest way to describe the protocol is to examine the connection setup process and the connection tear down process as we step through the protocol states on the server side of the connection.

Connection Setup

The server side of the connection starts in an IDLE state awaiting a Connection Request (CR) packet. The client side sends a CR packet and moves to the CONNECTING state. On receipt of the CR packet, the server sends a Connection Confirmation (CC) packet to the client and moves to the CONNECTED state. Once the client receives the CC packet, it also moves to the CONNECTED state.

Any of the packets can be lost in this interchange. It is up to the client to retransmit CR packets until either a timeout period expires or a CC packet is received. On the server side no timers will be started once the CR packet has been received and the server has moved to the CONNECTED state. This is consistent with the data transmission part of the protocol that does not use keep alive timers when no data is left to be acknowledged. This design makes the initiator of the connection responsible for driving the state transitions of its peer and reduces the number of packets exchanged. However, there is a flaw associated with this design. If a CR packet is sent to the server after which the client dies, there is no way to remove the state of the connection held by the server. In practice this situation rarely arises and when it does the amount of memory used to hold the protocol control block is small. More importantly, this design makes *slogin* extremely vulnerable to denial of service attacks similar to the SYN attack used against TCP servers such as WWW servers.

Connection Teardown

A connection teardown can be initiated by either side of the connection. The initiator of the teardown sends a Disconnect Request (DR) packet to the peer and moves to the DISCONNECTING state. On reception of the DR, the peer sends a Disconnect Confirmation (DC) packet to the initiator and moves to the idle state. As soon as the initiator receives the DC packet, it also moves to the IDLE state.

State Transitions

Table 2.1 shows the state transitions of the *slogin* protocol. The first column lists all the protocol packet types while the first row lists all the protocol states. Each table entry shows the protocol response to receiving a particular packet at a particular state. Each response entry consists of two rows. The first row shows the packet to be sent in response, if any. The second row shows the next state transition, if any.

For example, if an endpoint receives a DR packet while in the CONNECTED state, it will send a DC packet and move to the IDLE state. If it receives any subsequent DC packets while still in the IDLE state, it will continue sending DC packets but it will stay in the IDLE state.

2.5 Implementation Issues

2.5.1 Control packets

As stated before, the rlogin program uses TCP's urgent pointer (URG) to perform expedited data processing. Depending on the TCP implementation the urgent byte might not be delivered before the ordinary data. *Slogin* was originally designed to use Con-

STATES	IDLE	CONNECTING	CONNECTED	DISCONNECTING
PACKET TYPES				
CR	CC CONNECTED	RESET IDLE	CC	RESET IDLE
CC	RESET	CONNECTED		RESET IDLE
DATA	RESET	RESET IDLE		RESET IDLE
CONTROL	RESET	RESET IDLE		RESET IDLE
DR	DC	RESET IDLE	DC IDLE	RESET IDLE
DC		RESET IDLE	RESET IDLE	IDLE
RESET		IDLE	IDLE	IDLE

Table 2.1: *slogin* state transitions

trol Packets to deal with expediated data processing. Control Packets will carry special data and information that needs immediate delivery to the other end of the connection. Interrupt and flow control characters as well as information related to changes in the size of the user's xterm will be typically transmitted using Control Packets. However, the *slogin* usage of a send and wait protocol instead of a windowing protocol resolved the issue of expediated data processing without the need to implement Control Packets. An interrupt character is typically used to stop the current transaction. If an application is generating a lot of output in response to a user's command, it is necessary to stop the transmission of the remaining output upon the receipt of the interrup character. Typically in the TCP situation large amounts of data will be transmitted before the interrupt is processed. In the *slogin* case at most one network buffer will be sent after the interrupt character is sent. This is an interesting side effect of using a send and wait protocol instead of a windowing protocol.

2.5.2 Retransmission

Slogin uses packet retransmission to deal with packet loss. Whenever a packet is sent, the sender expects to receive an acknowledgement from its peer. If an acknowledgement does not arrive then either the original packet or its acknowledgement has been lost. To break the deadlock due to packet loss, the sender sets a timer once a packet has been sent. If the timer expires before an acknowledgement has been received, the packet will be retransmitted. To deal with the case when the peer disappears for whatever reason, a packet can be retransmitted for a specific number of times after which the connection will be closed.

To implement the above mechanism three parameters need to be specified:

- The value of the initial retransmission timer. This time is selected based on the round trip time. The exponentially weighted moving average (EWMA) from TCP is used.
- The value of the subsequesnt retransmission timers. These timers are exponentially backed off.
- The maximum number of retransmissions. This value is set to 11. A packet will be retransmitted 11 times after which the connection will be closed.

2.5.3 Bind and connect problems with UDP

A TCP connection is uniquely defined by a pair of identifiers each identifying one end of the connection. Each identifier contains a 4 octets IPv4 address and a 2 octets port number. Well known TCP services are available on well known ports. For example telnet uses TCP port 513. All telnet connections use port number of 513 at the server end and can use any port number at the client end. The addresses and ports are used to deliver packets to the correct process. In many systems the way this is implemented is to a have a process control block (PCB) with addresses (local and remote) as well as ports (local and remote). When a packet arrives at a host it is compared against the PCB's. It is delivered to the process which best matches the PCB entries. Both TCP and UDP packets contain source and destination addresses as well as source and destination ports.

Initially when a telnet server is waiting for a connection the local port number is bound to port 513 and the remote port number is unbound. There is no connection yet. As soon as a connection is created the remote port is bound to the port number the remote packet originated from.

In the TCP case this works well. Hence *slogin* used a similar method. The *slogin* server would wait on a well known port. The client would then ask the system for an currently unused port. The UDP packet would be sent to the server with a random source port and the known destination port. The packet would be delivered to the process which was listening on the *slogin* port. If the incoming packet was a connection request then the server process would start a new process. The new process would have a copy of the networking socket. The new process would then set the remote port value on its PCB using the connect(2) system call. Now both the parent and the child would have a PCB with the same local port. In the server case the remote port would be unbound. Thus all packets which are not bound to the child process will be sent to the parent. By the same token all packets for the child process will be delivered to it. Thus the operating system would be responsible for correctly demultiplexing incoming packets. Unfortunately on the Sun running (Solaris 2.5.1) under which I was writing this code the demultiplexing did not work correctly. The way that the bug manifested itself was that the first connection worked fine. If a second connection was attempted then no more packets were delivered to any of the applications. This seemed extremely surprising. One would have thought that other UDP based programs might use the same mechanism as was attempted in *slogin*. On checking the source for the Trivial File Transfer Protocol daemon(tftpd). A widely used UDP based file transfer program. It transpired that tftpd gets around this problem by sending the response packet from a different local port than the advertised port. Once the response is received by the client all packets are sent to the new port on the server. A similar scheme was therefore incorporated into *slogin*. An irritating feature of this scheme is that for every connection request a new process is created, even if the connection requests are coming from the same client.

2.5.4 User Privileges

The *slogin* program was developed on a Sun running Solaris 2.5.1. In order to test *slogin* on wide area links an account at INRIA ,in the South of France, was used. This account has no root privileges. It was therefore necessary to try and get the *slogin* server (slogind) running with only user privileges.

As soon as *slogin* manages a connection setup, including the Diffie-Hellman exchange, the server slogind runs the standard Unix login program to perform authentication.

Starting the login program turned out to be extremely problematic. The login program requires to be started from the lowest level shell. On UNIX systems a file “utmp” is used to hold login records. In order to start the login program the “utmp” file needs an appropriate entry. Trying to write the “utmp” entry as a non root process proved to be a problem. A mechanism was eventually found through a library. The “utmp” entry was not cleared on logout as the system will eventually perform this task. This causes a problem, for attempts to log in immediately after logout, the “utmp” entry isn’t cleared the same pseudo terminal is allocated and it is not possible to log in. The second login attempts seems to clear the entry and it is possible to log in.

2.5.5 Testing

The *slogin* protocol was tested initially on a local area network. For realistic wide area tests an account at INRIA was used as described above.

Also a system was built to emulate wide area lossy links. This system *flakeway* is described in Appendix B. *Flakeway* was table driven and hence deterministic. This allowed for repeatable tests to test the *slogin* protocol, using hand crafted data sets. Test data sets were also derived from ping traces between UCL and INRIA. It was initially hoped, to perform systematic tests of *slogin* against rlogin, to demonstrate the benefit of one scheme over the other. It was not clear that such tests would actually help in proving one scheme was better than another. Therefore the *flakeway* was only used for testing.

2.5.6 Security

At the time that the *slogin* work was done; 1994, there were no generally available remote login programs that attempted to cryptographically protect either passwords or session data. Since that time “SSH” [94] a secure login program has become widely available and is heavily used.

Any program being developed to support remote logins, really needed to support security. Adding security in the form of encryption also allowed for testing of ideas in integrated layer processing (ILP).

A major win with *slogin* is that as the protocol and application are so tightly coupled the connection setup exchanges can also carry pertinent data. Carried with the connection setup messages were Diffie-Hellman [22] setup information. Diffie-Hellman allows two parties that do not share a common secret to generate a common secret. It relies on the difficulty of factoring large numbers. In the *slogin* case the connection and Diffie-Hellman exchange can be taking place in the same packets. If TCP was used first a three way hand shake would need to take place, followed by the Diffie-Hellman exchange. More packets are exchanged than is necessary. Which is wasteful of bandwidth and time. Some transport protocols do allow data to be carried in the connection setup packets.

Once a secret key has been generated then the stream encryption protocol RC4 is used. RC4 is easy to implement and has no known attacks [78]. Once seeded by the key generated by the Diffie-Hellman exchange. RC4 generates a byte stream. This stream is exclusive or'd with the data stream. At the receiver the same RC4 stream is generated and the received data stream is again exclusive or'd with the incoming stream. This allows the recovery of the transmitted data. As well as being simple to implement it is also computationally cheap; this can be observed from the implementation in Appendix D. Another advantage of RC4 is that being stream based, unlike block solutions it is not necessary to pad transmitted packet to a block boundary, hence not necessitating the transport of extra padding bytes in the data payload.

The Diffie-Hellman key exchange uses a prime modulus of 1024 bits. The (prime - 1) / 2 is also prime. This is supposed to make the exchange more secure. I generated the prime myself which may have been a mistake. Its been tested against all primes less than 2000 and has had the Rabin Miller test run 5 times. A 1024 bit key is constructed from the exchange.

The stream cipher RC4 can accept up to 2048 bit key. So the 1024 bit key is repeated to give a 2048 bit key. In order to have a different encryption key in each direction the key is byte swapped for the reverse path. Hopefully this doesn't open a hole in the encryption.

The GNU multiple precision library was used to manipulate these large numbers.

The security in *slogin* is only there to encrypt the traffic. It does *not* perform any authentication. Authentication is performed by the standard login process. However, as all traffic is encrypted the password will not be available to the world. The Diffie-Hellman algorithm is susceptible to man in the middle attacks.

Checksum - Authentication

It is essential that all incoming packets are authenticated, otherwise bogus packets could be fed into the session. A nonce is added to the end of each packet before encryption. The nonce is a trivial checksum that is run across the data portion of the packet.

The checksum algorithm consists of summing all the bytes in the data field and then storing the low order byte at the end of the packet. This scheme is used for authentication, not for protecting the data. Before the payload is encrypted the checksum byte is placed at the end of the payload. When a packet is received it is decrypted and the checksum algorithm is run. Only if the computed checksum byte and the checksum byte in the packet match is the payload accepted. There is a very high likelihood that the creator of data packet knew the secret encryption key.

2.6 Experiments

On November 27 1994 the *slogin* and slogind programs existed. Experiments were performed between UCL in London and INRIA at Sophia Antipolis. The typical roundtrip time was around 200ms with 17 hops.

The original idea had been to build an asymmetric protocol, at the client side all data which had not been acknowledged would be sent in all packets, this seemed to be a fairly reasonable approach as at the client end, the input is from the keyboard. In order to simplify the implementation, the first cut was to build a symmetric protocol: unacknowledged data from the server was also repeated. With a terminal size of 24x80 this could mean 1920 bytes in a screen refresh, generated perhaps by an editor. However, experiments showed that the arbitrary packet size of 1024 which was chosen sometimes meant that a screen of data would require two packets. Host MTU discovery, would be extremely useful, to allow the application to transmit the largest possible packet size for a path.

In order to simplify the buffering of data both from the keyboard and the remote shell once 1024 bytes of data is in the transmit buffer the program no longer reads from the input keyboard/shell. This has the side effect of there only ever being one packets worth of data buffered. So if a keyboard interrupt is used to stop a flow, only one packet's worth of data is ever in the system. When a lot of data is being generated from the server end of the connection, for example displaying a very large file (e.g. cat /etc/termcap), the protocol becomes a send and wait protocol.

2.7 Results

On a long delay link with a small amount of packet loss the *slogin* program feels more responsive than rlogin. This is due to not having to wait for one round trip time when a packet is lost. It would be interesting to modify a TCP implementation to also send all buffered bytes when the number of bytes is small.

The XON/XOFF flow control and interrupt processing work very well even without special support, this is due to only ever buffering one packets worth of data.

As mentioned above when a lot of data is being sent the current implementation becomes a send and wait protocol. This means that currently *slogin* has a lower throughput than rlogin. However, the author believes that apart from performing tests, generating a lot of data which cannot possibly be read is not useful, so limiting the rate is not a bad property of the current implementation.

2.8 Enhancements

Passing the user name display variable and term variable would be useful The rlogin protocol also tracks window size changes. There is support for a separate control channel in *slogin* which has never been implemented. This control channel could be used for window updates.

The environment variables could just be tacked onto the end of the key in the key exchange. They could actually be placed in the packet as environment variables and not even require interpreting.

2.9 Discussion

TCP is badly matched to the requirements of a rlogin application. In most cases the application writer has information about the characteristics of his application that can be exploited. A classic example of this problem is rlogin, the echoing in rlogin is performed by the remote end, when it was discovered that for every character that was typed first an ack was sent back then the echo, TCP implementations were modified to delay for 200ms to delay the sending of the ack so that it could be carried with the echo.

We can make many observations about a typical rlogin session: a user does not type fast, so until an acknowledgement is received up to some threshold all characters typed can be repeated. On long delay paths this should give the best response under conditions of loss. Consider the case of a character typed by the user. The single packet carrying this character is lost. Then another character is typed. In the TCP case due to the Nagle delay typically 0.5 seconds will pass until the two characters are transmitted. In the *slogin* case the second character will be sent immediately with the first. If the TCP has Nagle disabled it will still take one more round trip time for the character to be delivered.

A user can not necessarily read very fast: data rate of the reverse channel could be kept to a fairly low settable rate. In fact as the protocol was implemented as a send and wait protocol there was always a RTT of delay. This provided a useful side effect of not requiring any special interrupt character or X-ON/X-OFF flow control processing. As there is only ever one window of data to interrupt.

2.10 Conclusions

Slogin provides a number of benefits over rlogin and telnet. It provides security at no significant overhead. This is also true of SSH.

Slogin provides better responsiveness on long delay links with any loss.

TCP is a good general reliable protocol, but it does not match the requirements, of even a simple application; such as remote login.

The actual *slogin* implementation was different to the original specification. The original specification was to have used windowing; the initial implementation used send and wait. This turned out to be perfectly acceptable. A benefit of this decision was, the control channel, which had also been specified, did not have to be implemented. The control channel was among other things supposed to carry interrupt characters. An expedited channel which could overtake the data channel. With send and wait there was never more than one packet of data which needed to be interrupted.

The control channel was also to have carried window size change information, amongst other things. Implementing this feature would not have provided any significant advantage.

It was harder to specify, implement and debug the *slogin* protocol than writing say rlogin. The author has written a rlogin for a commercial UNIX vendor, so can validly make this comparison.

The increased implementation complexity of this approach, would be an obstacle to its adoption. An avenue that was consider was the provision of building blocks to facilitate the construction of application level protocols (ALPs) such as *slogin* [32]. This direction was not pursued. The work in this chapter was done as part of the Hipparch project [5]. Partners in the project where looking at automatic protocol composition

[23], in order to solve the complexity of designing and implementing protocols.

In this chapter one application has been considered. In the next chapter we consider mailing lists to see if ALP concepts can be used in a very different scenario.

Chapter 3

MMD: Multicast Mail Delivery

3.1 Introduction

The work presented in this chapter is joint work with Zheng Wang and Jon Crowcroft. The work was done at the end of 1995, which is why some design decisions may not be as appropriate today as they were then. These decisions are discussed in section 3.4.4.

In this chapter we consider mailing lists as another application that could benefit from application level protocols (ALPs). This application is very different to the *slogin* application. It is again, however a very heavily used application.

As mailing lists grow large, current centralised distribution becomes increasingly unmanageable. In this chapter we present an alternative mailing list delivery system MMD. MMD delivers message to list members by multicast and achieves reliability with a lightweight retransmission scheme based on expanding ring local repairing. MMD can co-exist with conventional mailing list delivery and requires minimum changes to the systems.

Mailing lists are among the most useful applications on the Internet. It is impossible to make an accurate estimation as to how many mailing lists exist in the whole

Internet. As of June 2nd 1996, www.lisxt.com, a mailing list directory had a database of 37,416 lists.

Standard mailing list operation is fairly simple, a mailing list is set up at a site by allocating an electronic mail address, for example *mmd@cs.ucl.ac.uk*. All electronic mail sent to this address is then fanned out to all members of the list at the main list exploder.

There are a number of problems associated with this conventional mailing list operation [89]. First of all, the administration of large mailing lists can be a very time consuming activity. The simplest part of maintenance may just be dealing with subscribe/unsubscribe messages. Although mailing list software typically can handle automatic subscription and unsubscription [81], people often fail to change their mail addresses in the lists when they change organisations. A major problem reported by maintainers of large mailing lists is delivery failure. When a mail transfer agent [75] fails to deliver mail to a recipient of a mailing list, a mail message is often generated by the mailer and sent back to the list maintainer with an error report. There can be many reasons for delivery to fail, for example, the hostname cannot be found in the DNS [60], the mailbox may no longer exist, the machine that the mailbox resides on has been unavailable for prolonged periods etc . . . Some of the problems may be the kind of transient errors that occur in networking or they may be more permanent like a recipient has changed organisations and no mail forwarding is enabled to the new organisations. Whether the error is transient or permanent, time is spent by list maintainers processing the delivery reports. For example, Information Sciences Institute (ISI), which maintains mailing lists for Internet research groups, the IETF and other Internet groups, on average received about 400 messages a month requesting additions or deletions, and about 300 error messages a month due to mail delivery problems during 1991 [89]. The members on most ISI's mailing lists have increased substantially since.

Since every message has to be replicated at the mail server where the mailing list resides, a large and active mailing list may place significant workload on the mail server and also outgoing traffic for the list maintainer's site. The majority of the mailing lists on the Internet are operated as a free service to a particular community or interest group, so it is desirable to reduce the burden to the list maintainer's site as much as possible.

When the number of members in a list reaches a few thousand, such centralised distribution becomes increasingly unmanageable. In this chapter, we present a distributed approach based on multicast, called *multicast mail delivery (MMD)*. In MMD, mailing list distribution utilises the multicast capacity of the underlying networks (MBONE)[17]. Messages from the main exploder are multicast out and each site interested in a particular list has a local exploder which listens at the multicast address and distributes to local members of the list.

With this approach, the entire mailing list operation is fully distributed. The administrative load on the main exploders is drastically reduced. Subscribing/unsubscribing is a local matter so there will no longer be the large turn-a-round time often encountered with some mailing lists. Any errors generated are handled locally and hence easier to solve. As the replication is done with multicast, the traffic load on the main exploders is no longer an issue. With local retransmission mechanisms, main exploders only need multicast a message once, or at most a small number of times when local repairing fails. Thus, MMD can deliver mailing lists more efficiently in terms of network bandwidth and speed of delivery. With MMD, there is the additional local burden of configuring a new local list each time a new mailing list is joined. This is inevitable as a result of the distributed approach. However, the setup process could be automated.

3.2 Design Basics

MMD was developed as an alternative to conventional mailing list delivery. Its aim is to relieve the administrative burdens on list managers, and to reduce traffic and delay in message distribution. The core of MMD is a multicast based distribution system with a lightweight retransmission scheme based on expanding ring local repairing. The design of MMD adheres to the following basic principles:

1. Co-existence with conventional mailing list delivery. Given the large installed base of conventional mailing lists, any systems incompatible with the conventional one will not work in practice. The use of MMD and conventional mail delivery must not be mutually exclusive. For any list, users who do not have access to networks with multicast capacity can continue to use conventional mailing list delivery.

2. Minimum modifications to current systems. MMD was designed to replace only the replication part of the mailing list delivery. There has deliberately been no attempt made to replace current mailing list maintenance software. Subscribing, unsubscribing, mailing to a list and receiving mail are handled through existing infrastructure. By doing so, MMD requires minimum changes to the current systems.

3. Handling of multiple lists. MMD was designed to handle multiple mailing lists so that one invocation of the program and configuration file would be required

per site. It will also allow multiple lists to share a simple multicast group.

3.3 The Structure

The MMD suite of programs is broken into three distinct components:

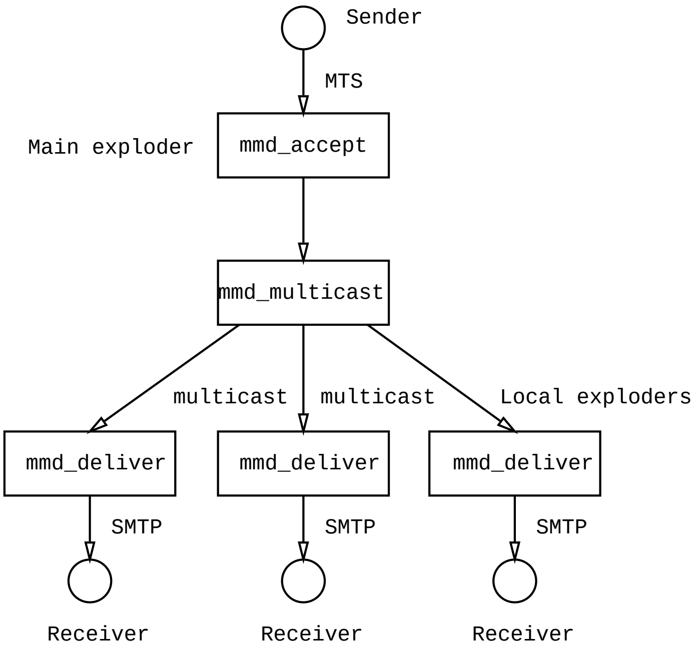


Figure 3.1: The Structure of MMD

1. mmd_accept

   Electronic mail enters the MMD subsystem via mmd_accept. The program is invoked by the mail delivery program, it then makes a unicast connection to the mmd_multicast program which is responsible for multicasting the mail messages. An example of operation would be set up a mailing list for example *mmd@cs.ucl.ac.uk*, rather than the mail now going to multiple recipients it is just sent to the mmd_accept. The mmd_accept program is a trivial program which is the entry point into the MMD subsystem. This program is invoked once per mail message and does little more that open a connection to mmd_multicast and relay the message. In the unlikely event of mmd_multicast not being available mmd_accept returns an error status and the mail message is queued by the mail software for a later delivery attempt as mail transfer systems are good at this. All retries are left to the existing mail infrastructure.

2. mmd_multicast

   This program mmd_multicast is a server process which is intended to run continuously, and is responsible for multicasting the mail message. This program sends out periodic status messages containing information about what messages are currently available. This allows receivers mmd_deliver to send negative acknowledgements when they discover that new messages are available.

3. mmd_deliver

   The program mmd_deliver is a server process which is responsible for reliably receiving the multicast mail messages and performing local delivery. It can also perform retransmissions in the event of loss.

3.4 The Protocol

The protocol is totally text based as are mail messages. Two channels are used, one for control messages and the other for the data. There were two reasons for this choice, one for ease of de-multiplexing of messages, the other to allow receivers to leave the multicast group on which the data is sent in order to take full advantage of multicast pruning, so once a site has received all the currently available mail it no longer needs to receive the mail again.

3.4.1 Status message

The message consists of colon separated fields:

v1:STATUS:timeout range:list name:lowest mail message:highest mail message:

1. v1

   The version of the protocol. In this case version 1.

2. STATUS

   The type of the message. The keyword STATUS is actually in the message.

3. timeout range

   In order to guard against the ACK/NACK implosion [27] problem a timeout range in seconds is sent with the message. Any receivers wishing to send a NACK in response to a STATUS message chooses a random time value within this range.

4. list name

   The name of the list.

5. lowest mail message

   The mail messages are numbered, this is is the lowest mail message available.

6. highest mail message

   The highest mail message available.

3.4.2 NACK message

The message consists of colon separated fields:

v1:NACK:list name:mail message:<optional fragment number or range>:

1. v1

   The version of the protocol. In this case version 1.

2. NACK

   The type of the message. The keyword “NACK” is actually in the message.

3. list name

   The name of the list.

4. mail message

   The mail message that is being NACK’d.

5. <optional fragment number or range>:

   If this field is not present then the whole message is being NACK’d. A single value denotes a particular fragment is being NACK’d. A range of fragments can be specified by giving a start value and an end value separated with an “-”, e.g. “10-15”.

3.4.3 Data message

The mail messages are broken into fragments. The sender can choose any appropriate fragment size, the only stipulation being that once a size is chosen it must be the same for all fragments and must remain constant for a particular message.

Each mail message fragment is prepended with the following header. The fields in the header are colon separated and the last entry is followed by two newlines (ASCII nl, hex value 0xa), after which the mail fragment appears.

v1:list name:message number:fragment:total number of fragments:

1. v1

   The version of the protocol. In this case version 1.

2. list name

   The name of the list.

3. message number

   The mail message number.

4. fragment

   The fragment number of the message being carried.

5. total number of fragments

   The total number of fragments which make up this mail message.

3.4.4 Retransmission Scheme

In MMD, retransmission is receiver-initiated based on expanding ring local repairing. When a receiver detects that a message is lost, it first requests the message in its local scope. Any entity which has a copy of the message can reply to such a retransmission request. If the requested message is not retransmitted, a receiver can increase the scope of its retransmission request until the requested message is received. We now describe the details of protocol operations.

The "mmd_multicast" program sends out periodic "STATUS" messages. The messages themselves are simple text messages. The "STATUS" messages carry information regarding the range of messages available. If a "mmd_deliver" program detects that it does not have a particular message it can send a "NACK" message in response to a "STATUS" message.

In order to guard against a “NACK” implosion, the “NACK” messages are not sent immediately after receiving the “STATUS” messages. If there were many receivers, sending “NACK” messages immediately after a “STATUS” message would lead to many “NACK” messages being sent at roughly the same time, thus potentially causing heavy congestion. Ideally only one receiver should send a “NACK” in a particular time period. A protocol could have been developed in order to synchronise all the participants to guard against the “NACK” implosion problem. This would however have increased the complexity. Also the number of messages exchanged would have increased.

As the distribution of mail makes no real time guarantees, a simpler strategy was adopted, using a random element to determine when a “NACK” should be sent. The “STATUS” messages carry a timeout value which is a interval within which a random value is chosen by each receiver. This value is used as a delay period from the arrival of the “STATUS” message. When this delay period passes, the receiver may send a “NACK” message. In the intervening period another receiver may already have sent a “NACK” for the requested message. As all messages are multicast, all receivers will receive the “NACK” messages, so a list of “NACK” messages is recorded and only new “NACK” messages sent in any particular period. In order to grossly simplify the present implementation if *any* “NACK” or “DATA” messages are seen in the delay period then no “NACK” will be sent, a new delay period will be chosen and the process will repeated until there are either no outstanding messages or there has been a quiet delay period when a “NACK” can be sent.

At this point some background is required. At the time that this work was done the MBONE was based on a distance vector multicast routing protocol (DVMRP). Since that time the protocols used have changed. What multicast protocol is currently in use does not affect our discussion. An important feature that has changed is how far a multicast packet travels in the network is controlled. How is the scope of a multicast packet controlled? The initial MBONE consisted of tunnels between participating sites. The tunnels were configured with thresholds. For a packet to traverse a tunnel its time to live (TTL) had to be larger than the tunnel threshold. Therefore how far a multicast packet travelled was governed by its initial TTL. The higher the initial TTL the further the scope of the packet. For example at the time this work was done a TTL of less than 31 would keep a multicast packet within the UK. A TTL of less than 16 would keep a packet within UCL. This was not considered to be a flexible enough scheme. Today multicast packets can be bounded by using administrative scopes [58]. Rather than using TTL's to bound the scope of a packet the multicast address is used. To greatly simplify the explanation, border routers in administrative domains do not allow packets with certain administrative address ranges to traverse them.

The retransmission is carried out in a distributed fashion. in MMD, not only can the "mmd_multicast" program respond to a "DATA" message but so can the "mmd_deliver" programs. An expanding ring search for local retransmission is used. The TTL was increased for "NACK" messages to expand their scope. Therefore topologically close sites had the opportunity to respond first. If there were no responses the scope of the search could be increased, until the whole world was searched. This TTL based expanding ring search would not work today as explained above. Today it would be necessary to configure a list of addresses that would be sequentially searched. Each address would have a greater scope than the previous entry.

The "NACK" messages are multicast, therefore all entities that have a copy of the message are in the position to reply with the requested “DATA”. The hope is that in all but the initial distribution of a message, the retransmission request will be satisfied by an entity closer than the original sender. As with “NACK” messages the problem still exists that a large number of “DATA” messages could be sent in response to a “NACK” message. Causing a “DATA” implosion problem.

As with “NACK” messages “DATA” messages are sent after a random delay in order to guard against implosion and synchronisation problems. All entities have a queue of outgoing messages. Whenever a “NACK” message is seen all entities which have a copy of the message place it on their outgoing message queue. The “DATA” may already be on the outgoing message queue, in which case it will not be added. At some random time in the future the outgoing message queue will be activated and the queued “DATA” packets will be transmitted. Every received “DATA” packet is compared against those in the outgoing queue, any that match are removed. If an entity requests a “DATA” packet by sending a “NACK”, many entities may be in a position to service this request. All sites with the appropriate “DATA” will schedule a transmission in the future. As soon as the first site satisfies the request the other sites will remove the transmission from their outgoing message queues.

The “STATUS” message carries a timeout value so that the value can be varied as the size of the group changes. Currently the timeout value is a small fixed value (60 seconds). The intention is that the “mmd_deliver” program should notice if many “NACK” messages appear together. If many “NACK” messages appear together, this signals that the timeout value is too low and the timeout value can be increased. When a new “DATA” message becomes available and no “NACK” messages are received in say half the timeout interval then the timeout value can be gradually decreased. The

"DATA" messages are never sent out unsolicited, only when a "NACK" is received will a "DATA" message be sent. It would seem more obvious to multicast all "DATA" messages as soon as they are received. The main reason for always soliciting a "NACK" is that it enables a more frequent estimation of the size of the group.

3.5 Related Work

A simple transport protocol called *Muse* for multicasting USENET newsgroups was presented in [51]. Muse distributes each news article in a single UDP multicast packet and there are no retransmission mechanisms for loss recovery.

There have been a number of proposals for reliable multicast transport in the literature [27, 18, 92, 8, 57]. Many use a centralized distribution tree for totally ordered multicast delivery. A reliable multicast scheme based on distributed loss recovery called *SRM* was discussed in [27]. The retransmission scheme used in MMD is similar to that in SRM where repair for lost packets are done locally.

3.6 Discussion

In the MMD design, we have assumed that the mailing lists are open to the public, thus no security mechanisms were incorporated. For closed mailing lists, conventional mailing list schemes can be used or the mail can be encrypted to maintain secrecy. At the present time with the prevalence of SPAM it would be necessary to use an authentication scheme to stop the unauthorised insertion of messages.

The MMD suite of programs was implemented in C++ under Solaris and SunOS. The implementation was tested for six months from December 1995 on a subset of the "rem-conf" mailing list.

3.7 Conclusions

One potential failing with the scheme described is that of scaling, from the perspective of an individual mailing list with many subscribers. If a mailing list has a thousand subscribers distributed through the world, then in the best case, a small mail message could be be transmitted wide area with a single multicast packet transmission from the sender. At the local sites there will of course be TCP connections to perform the actual mail delivery. In conditions of loss there will of course be retransmissions. This would be true of TCP as well. There is also the issue of the low frequency status messages. These may generate more traffic than TCP connections on low volume mailing lists. For individual mailing lists the benefit of MMD can be seen. The real scaling problem arises in the case of large numbers of mailing lists. Although the protocol was designed such that information for multiple lists could be carried together; this would not scale to carrying all the world's email list. One possible solution may be to use many multiple multicast addresses. This part of the problem space has not been addressed.

As with the *slogin* work presented in the previous chapter an ALP has been developed and shown to work in a small scale test. The resources were not available for a truly global deployment and test. Nodes were run in London, Cambridge and Edinburgh.

We are able to show as in *slogin* that we can design our protocol to take advantage of our knowledge of the protocol. In the MMD case there is no timeliness requirement. Mailing lists are not required to provide real time delivery. A low frequency status message can keep receivers apprised of the current messages available.

Two very different applications have been chosen and ALPs have been built to demonstrate that there is benefit from using this approach. They are provided as a proof of concept, as a basis for the rest of the work presented in this thesis.

There is still however the issue of composing ALPs and deploying them. As has already been stated other members of the HIPPARCH project were looking into automatic protocol composition [23]. Some initial work was also done in providing building blocks for protocols composition [32], this avenue was not pursued.

In the rest of this thesis our emphasis will be on the deployment aspect of the problem. Entities called *proxylets* are described which allow code to be deployed in the “network”. Such a scheme could be used to deploy a MMD into the network. An additional benefit of such an approach would be that application layer multicast could be used. Therefore no reliance on the current variation of the available multicast technology. These ideas will be explored in later chapters.

The next step in the path towards ALP deployment is presented in the next chapter.

Chapter 4

JavaRadio: an application level active network

4.1 Introduction

The work presented in this chapter is joint work with Michael Fry [35].

In the previous chapters (2, 3), we looked at the benefits of Application Layer Protocols (ALPs). In this chapter we describe an experiment, which is the first component in building an application level active network. ALPs have benefits but they are harder to develop than applications built on, for example TCP. If ALP's could be made portable, hence easier to deploy they would become more attractive. The extra effort in development could be counterbalanced with ubiquitous deployment. The ALPs and the protocol server idea was not the approach that was ultimately adopted. This chapter is left in this form to show the gradual evolution of ideas.

The World Wide Web (WWW) [10] is the world's largest distributed database. Large amounts of data are available in various formats. However, we believe that the Hypertext Transfer Protocol (HTTP) [25] over Transmission Control Protocol (TCP)

[71] is not always the most appropriate mechanism for retrieving this data. An obvious example is audio files stored in the WWW. Typically, to access an audio sample one has to download the whole sample and then listen to it. A four minute audio sample at 8K bytes per second would be 1920 Kbytes. If there is a slow or lossy path between the receiver and transmitter, the download could take a long time (e.g. from a WWW server at University College London to a client at the University of Technology, Sydney usually took about three and a half minutes in 1997).

Playing out the sample as it arrived would seem a more appropriate solution. However, streaming audio over TCP can cause problems if there is any loss. Under conditions of loss a delay builds up as lost packets are retransmitted. A Real Time Protocol (RTP) [80] [79] stream would seem more appropriate as some packet loss can be tolerated. A possible solution for listening to audio streams may be to modify the HTTP protocol to return content using RTP streams. This solution is limited as every WWW server would have to support many protocols.

Audio files are just one example of files available on the WWW where the standard HTTP transactions over TCP are inappropriate. Another solution widely in use is RealAudio [64]. In this case a URL refers to a dedicated audio server which streams out audio using a protocol appropriate for audio. The audio is held on a separate server, so can only be accessed with dedicated tools. We propose a more generic solution where the actual content such as audio continues to reside on a WWW server.

4.2 Proposal for Mobile Protocol Stacks

We propose initially that proxy servers exist "close" to the WWW server from which content is required. A standard URL is sent to the proxy along with a either an Application Layer Protocol (ALP) or a reference to an ALP. The proxy then extracts the content from the WWW server and returns it using the ALP. This allows the client to select how content is returned. The client in effect causes the pushing of the playout protocol to the proxy which is close to the WWW server. Ultimately it may make more sense to have the proxy co-resident with the WWW server.

The client may also pull an ALP in order to "view" the content if it doesn't already contain an appropriate playout mechanism. Repositories of ALPs (protocol servers) could exist which contained both ends of an ALP stack. Possible protocol version mismatches would be solved by having both ends of the an ALP stack located together. An application would be at liberty to use any ALP available and signal to the proxy which playout ALP to use.

To achieve this we require an infrastructure that supports mobile protocol stacks. Although the protocol server is described here in the context of the WWW it may be used by other applications to upgrade a protocol stack. In the initial work that we describe the protocol server has not yet been implemented but a proxy with fixed protocols and a client have been built.

4.2.1 Application Choice

To test these ideas we wanted to build an application with various configurable protocol options. An initial application was required which could be implemented with a number of different protocol stacks. An audio application was chosen for several reasons: it could be implemented with separate protocol stacks, the various audio coding schemes add another dimension in variability, the real time aspect tests the Java implementation framework we have chosen (below), it is easy to demonstrate, and it is a contemporary application.

We have now built a framework using the Java programming language to begin experiments with mobile protocol stacks. In order to support mobile protocol stacks it is desirable that the stack is portable and not platform specific. A simple way of meeting this criterion is to use Java. Java programs are compiled to byte codes which are interpreted by a virtual machine. Java programs will therefore run on all architectures for which an interpreter exists. However interpreted code will not execute as fast as native code. Just in Time (JIT) compilers are becoming available for many platforms. A simplistic view of a JIT compiler is that it converts the portable byte codes to native code to remove the performance penalty of interpreted code.

Although Java offers portability it is not at all clear if it is suitable for building protocol stacks. For example we show that header processing in Java is more complicated than in the C [46] programming language, which is the most common language currently in use for protocol building. Another unknown was whether Java delivers the performance required for the current generation of applications.

A standard for transmitting real time audio exists. We decided to use the Real-time Transport Protocol (RTP) as one of our configurable protocols. Another protocol which is supported is TCP. As well as a playout tool it was also necessary to write a transmitter which could take audio files and generate RTP or TCP audio streams. To enable protocol and codec selection a control tool was built to interact with the proxy. The control tool is used to select various aspects of the playout such as protocol stack and codec selection.

4.2.2 Significance

We are attempting to build a framework for the deployment of new protocols rather than just build a new playout tool. The work described currently uses fixed ALPs. We intend to move towards dynamically deployed ALPs. An example of a simple ALP that we have in mind is an ALP which compresses WWW pages. This would be downloaded to a remote proxy which would compress a page before transmission. The decompresser could be pulled to the local client. This is a compressor/decompresser ALP to be used in saving transmission bandwidth. The benefit is that the playout protocol and transformation on the data is determined dynamically at runtime.

This work is also a contribution to research into Active Networks. We discuss this aspect below.

4.3 Experiment Description

The structure of our experimental framework and application is shown in Figure 4.1. Rather than modify a WWW server to support downloading of protocols, we have written a separate proxy server which is placed "close" to the target web server.

An audio tool: Yet Another Audio Tool (YAAT), waits for incoming audio streams and plays them out. The proxy server accepts requests for audio URLs. The audio sample referenced is downloaded to the proxy server and then played out to the audio tool. The proxy server accepts requests through a Java RMI interface. The requests are sent to the proxy server from the control interface using the RMI interface. Although a Protocol server is shown in the figure it has not yet been implemented. For our wide area experiments a proxy server was located at Martlesham Heath in the UK and the playout was performed to Sydney in Australia.

The proxy server accepts a series of parameters: a URL for the audio sample, a stream type RTP+UDP or TCP, a codec to use (PCM or ADPCM), an audio sample size (20ms, 40ms, 60ms and 80ms), a redundancy level (up to two levels) and a destination for the stream. These are described further below.

The interactions are as follows. A Universal Resource Locater (URL) is sent to the proxy server of an audio sample on the WWW server. While the audio sample is being pulled to the proxy server the playout is started with the parameters above. If network conditions change, such as high levels of loss, the playout can be modified. For example we can select a higher level of compression, such as changing from PCM to ADPCM compression, which gives a 50% saving in bandwidth. If there is high loss rate, redundancy can be added to each packet by carrying the previous audio sample in the current packet. Currently two levels of redundancy are supported. At present the various parameters can be modified by the user. However it would be relatively simple to add a Quality of Service manager which, as a function of feedback at the receiver, modified the audio stream to attempt to bring it back to acceptable quality.

The code has been written in Java, which is compiled to byte codes which are then interpreted inside a Java Virtual Machine. This means that we have no dependency on a particular machine architecture. A performance penalty may be paid for using interpreted code.

With the portable byte codes that Java compilers generate, protocol stacks could be written in Java and then exported on demand to their point of use. This is discussed below under future work plans. Working with Java has also allowed us to evaluate the problems of writing protocol code in Java, as well as discovering any potential performance problems when using Java for a real time application.

At present our infrastructure does not yet support code mobility. We do have RTP compliant Java code, two codecs: PCM and ADPCM, an audio playout tool and a proxy server. These are now described.

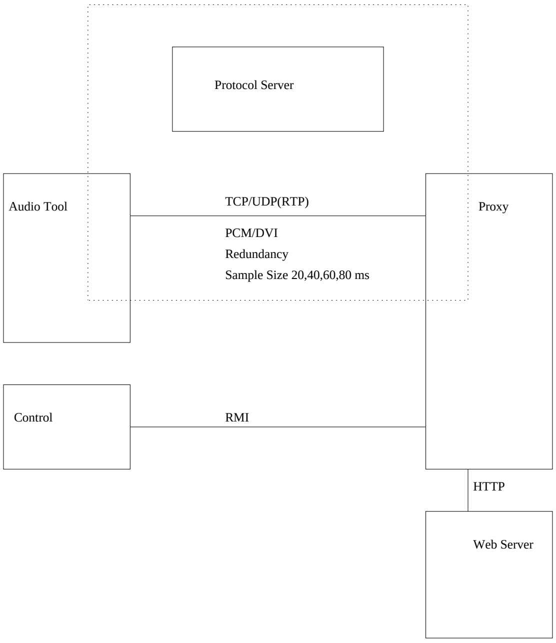


Figure 4.1: Framework

4.4 Component Overview

4.4.1 YAAT - Yet Another Audio Tool

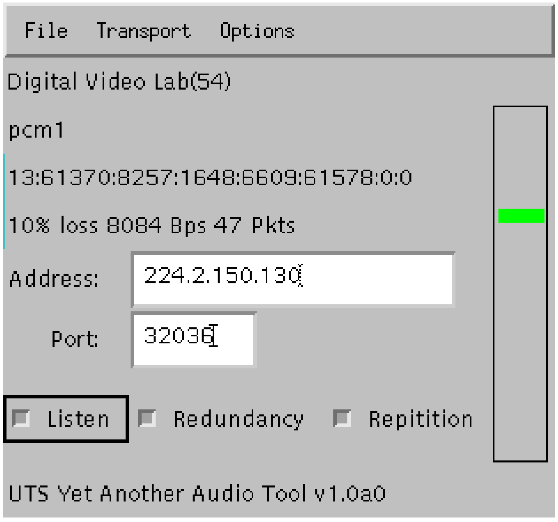
The audio tool YAAT is totally written in Java (Figure 4.2). Three methods of playing out streams are supported.

1. TCP download

   In this mode a TCP connection is used to download the audio sample. The whole audio file is downloaded and then played. A delay is incurred while the file is downloaded. The user has to wait for the whole file to be downloaded but the audio quality is preserved.

2. TCP streaming

   In this mode a TCP connection is used to download the audio sample. The audio playout is started after a buffer of playout data has been gathered. In this mode the user can start to listen to the audio sooner. Every time a packet is lost, at



Digital Video Lab(54)

pcm1

13:61370:8257:1648:6609:61578:0:0

10% loss 8084 Bps 47 Pkts

Address: 224.2.150.130

Port: 32036

UTS Yet Another Audio Tool v1.0a0

Figure 4.2: Yet Another Audio Tool

best a delay of one round trip time will be incurred at the receiver. So for every packet lost there is a build up of delay for the transmitted packets. If the playout buffer is not large enough, eventually the receiver will have no data. If a no data condition arises, then the audio will break up as there will be periods of silence. The larger the audio sample then the larger the playout buffer has to be, due to the potential for the build up of delay.

3. RTP streaming

In this mode RTP is used over UDP. Various options are available to the user via the control interface for the playout. Options can be varied during playout.

The options available are:

- Codec

  Currently two codecs are available. The PCM codec actually does nothing, as the audio samples in the file at the server are assumed to be PCM. The audio device is also assumed to accept PCM. The other codec is an ADPCM codec which gives a compressed sample half the size of the original data sample. We were unable to perceive a change in the quality [21] of the audio when switching from PCM to ADPCM.

- Sample Size

  The length of each audio sample can be varied from 20ms to 80ms in 20ms increments. Obviously changing the sample size changes the packet size.

- Redundancy

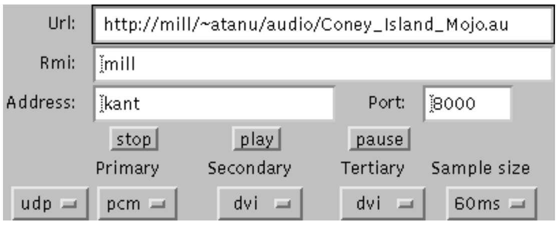
  A redundant packet format is supported. The current payload may carry a packet from the previous sample.

- Response to loss

  In the event of packet loss no audio sample is available to place in the audio playout buffer. In this case rather than insert a 20ms silence sample, the previous audio sample is repeated [37]. This repetition greatly enhances the quality of the audio.

4.4.2 Proxy Server

The proxy server supports a RMI interface for accepting requests. It also supports HTTP to retrieve audio files. The proxy server transmits and possibly converts the retrieved file either over a TCP connection or over a UDP+RTP stream. The proxy supports audio encoding in PCM and ADPCM. The sample size of the transmitted packets can be varied from 20ms to 80ms in 20ms increments. The proxy can also add redundancy. The redundancy works by adding an audio sample from the previous packet to the current packet. A packet contains the *n* sample and the *n-1* sample, perhaps using a different codec. The scheme is not limited to just one level of redundancy. In fact the server supports placing two levels of redundancy in each packet. So a packet can



Url: http://mill/~atanu/audio/Coney_Island_Mojo.au

Rmi: mill

Address: kant

Port: 8000

udp

Primary: pcm

Secondary: dvi

Tertiary: dvi

Sample size: 60ms

Figure 4.3: Control Interface

contain the *n*, the *n-1* and the *n-2* sample.

4.4.3 Control Interface

The control interface (Figure 4.3) is used to control the playout of the audio stream. The various parameters such as the codec the sample size and the levels of redundancy can be modified while the stream is being played.

4.4.4 Protocol Server

The protocol server is currently not implemented, but ultimately the codecs and the RTP components of the protocol stack will reside on it. The intention is that both server and client components of a protocol are developed and placed on the protocol server. This gives the ability at run time for both ends of a communication to adopt a new stack. The new stack may be adopted for a variety of reasons: performance enhancements, bug fixes, new stacks, etc. The idea is that the protocol server will allow the rapid deployment of new protocols. Protocol engines often have to support old versions of a protocol for backward compatibility. In a world with protocol servers if a protocol mismatch occurs then one or both sides of a communication can adopt a new stack.

4.5 Implementation Issues

4.5.1 Portability

All the code was written in Java in order to have portable code which could run on any platform. One exception had to be made however. Java is a fairly new language and all the APIs are not currently implemented or deployed. No portable way exists of accessing an audio device from Java. So unfortunately YAAT will run on UNIX workstations where it makes the (possibly very poor) assumption that the file "/dev/audio" will give access to the audio device. With time a portable way of accessing the audio device should become available.

4.5.2 Stability

In the few months that the work has been underway Java has undergone a series of changes. In most cases the old APIs have been preserved so, though no necessity existed for changing code, if new functionality was to be used changes to the code were necessary. A very irritating feature of Java is that is does not support conditional compilation. The thinking behind this decision is that Java has no architectural dependencies so there is no need for conditional compilation. As the APIs evolve it is however often useful to be able to build code to the old API. An example of the problem is that in one of the JDK releases support for IP multicast was introduced into the networking API. This support was integrated into YAAT (This enabled YAAT to be used to listen to MBONE sessions). Unfortunately in this release of the JDK the graphics performance was seriously broken. It was necessary to support two versions of the code in order to perform benchmarks. Conditional compilation would be extremely useful in these situations.

4.5.3 Threads

Many features of the Java language make it easier to use than more conventional languages. One of the best features of the language is that thread support is built into the language. A common problem in using a useful feature like threads in other languages is the worry that, if the code is to be truly portable, a threads package may not be available on another platform. With threads as part of the language they can be used with the knowledge that they will always be available.

Thus with threads and the object oriented nature of Java, it is very simple to partition tasks into separate threads. For example in YAAT separate threads are responsible for the audio device, networking and the windowing interface. The threads can support different priorities, so the audio thread has the highest priority of all the threads in YAAT. The very nature of networking applications means that network input events are asynchronous. A real benefit of threads is that a separate thread can be allocated to the various network input events.

Our RTP implementation has data packets on one stream and RTCP session packets on another stream. All the session packet construction and reception would be common across all applications. Threads allow the building of a reusable RTP library which uses a number of threads to support the session and data messages.

4.5.4 Header Processing

An application which is doing its own Application Layer Framing (ALF) needs to process and create header fields in packets. In YAAT incoming RTP and RTCP needs to be processed. In the proxy server, RTP and RTCP packets have to be created from the audio stream for transmission.

```
struct rtphdr {
#if     defined(sparc) || defined(hpux)
        unsigned char    rtp_v:2;       /* Version */
        unsigned char    rtp_p:1;       /* Padding */
        unsigned char    rtp_x:1;       /* Extension */
        unsigned char    rtp_cc:4;      /* CSRC count */

        unsigned char    rtp_m:1;       /* marker */
        unsigned char    rtp_pt:7;      /* payload */
#endif
#if     defined(vax) || defined(ALPHA)
        unsigned char    rtp_cc:4;      /* CSRC count */
        unsigned char    rtp_x:1;       /* Extension */
        unsigned char    rtp_p:1;       /* Padding */
        unsigned char    rtp_v:2;       /* Version */

        unsigned char    rtp_pt:7;      /* payload */
        unsigned char    rtp_m:1;       /* marker */
#endif

        unsigned short   rtp_seq;       /* sequence number */
        unsigned int     rtp_time;      /* time stamp */
        unsigned int     rtp_ssrc;      /* Synchronization source identifier */
};
```

Figure 4.4: RTP header

```
struct rtphdr *rtp;
char buf[ENOUGH];
read(net, buf, ENOUGH);
rtp = (struct rtphdr *)buf;
seq = rtp_v2->rtp_seq;
```

Figure 4.5: Header Processing in C

No simple way exists in Java of extracting fields from a header or for putting fields in. In the C programming language, the most commonly used language for this style of processing, a structure is declared with the fields in the header. An example for a RTP header file is shown in Figure 4.4. In C this structure would be cast over the incoming bytes and the fields would then be accessible. A simple example is given in Figure 4.5.

In Java no such idiom exists and it is necessary to extract each field individually. A simple example is given in Figure 4.6. Two problems exist with Java header processing.

1. Each field has to be individually extracted. In the C case as shown, a simple structure is declared as an overlay on the header fields. If a new field is added or removed it is a simple addition or removal from the structure. In the Java case a

```
...
byte b[];
seq = Conv.byte2_to_int(b, 2);
...

public static int byte2_to_int(byte b[], int offset) {
        int i;
        int val = 0;
        int bits = 16;
        int tval;

        for(i = 0; i < 2; i++) {
                bits -= 8;
                tval = b[offset + i];
                tval = tval < 0 ? 256 + tval : tval;
                val |= tval << bits;
                }

              return val;
}
```

Figure 4.6: Header Processing in Java

lot more care has to be taken in order extract the correct fields.

2. All integral quantities in Java are signed. The sequence number is a 16 bit (i.e. two byte) field. So two bytes have to be extracted from the packet to form the sequence number. As all integral quantities are signed all byte values greater than 127 will be sign extended and be negative. So the extraction code for each byte has to check if the value is negative and then convert it back to a positive value. This is extremely inefficient. Every multibyte field must have all bytes checked to guard against sign extension.

One interesting distinction between C and Java in the context of the header processing is that, as Java defines the size of all its base quantities, the Java code once written will always work. The same can not be said of the C code, where the size of integer quantities are not defined (Compiler Dependent). Also, as can be seen from figure 4.5, the ordering of the fields in a structure is not defined.

4.6 Experimental Outcomes

The most important question which we wished to answer was: could Java be used for building ALPs? This question can be broken into two subsidiary questions.

1. Is the Java Virtual Machine (VM) fast enough to allow the running of real time code, such as the audio tool and the playout code in the proxy?

2. Does the Java language and API provide reasonable hooks for writing protocol code?

An initial experiment was performed locally at the University of Technology, Sydney (UTS). The WWW server and the proxy were on the same machine and the control interface and YAAT were run on a machine on the same subnet. Both machines involved in the experiment were Sun Ultra 1's, running Solaris 2.5.1. A three minute song was used as the test sample.

The main computational concern was that an audio sample could be coded or decoded in less time than the sample represented. The smallest time sample supported is a 20ms sample. For PCM audio samples the codec does nothing as the file contains PCM samples and the Sun audio device accepts PCM. In the ADPCM case the codecs convert from PCM to ADPCM or ADPCM to PCM. To decode a 20ms ADPCM audio sample on a Sun Ultra 1 takes, between 1-2ms. This is well inside the 20ms required. The worst case is the proxy where 20ms ADPCM audio samples are created with two levels of audio samples. This process takes 4-5ms. The time taken by the proxy for two levels of redundancy for 20ms samples and the transmission of the packets including the packetisation is $\sim$5ms. So using the ADPCM codec with two levels of redundancy and a sample size of 20ms a packet needs to be transmitted every 20ms. It actually takes ∼5ms, which leaves ∼15ms available. So for a simple codec like ADPCM there is plenty of time.

4.6.1 Local Area Experiment

In order to simulate the effects of network loss, jitter and reordering, the RTP and RTCP packets were passed though a *flakeway* (Appendix B) which induces loss and jitter. One of the *flakeway* configurations tried was to drop 2/3 of the packets passed through it. For every three packets arriving at the *flakeway* only one was allowed to pass (the other two were dropped). This was a pathological case which enabled the testing of the redundancy code. With two levels of redundancy enabled the dropped packets appeared in the redundant part of the payload of the surviving packet. The whole stream could therefore be reconstructed. With this high level of loss which was distributed favorably for the application the received sound quality was acceptable.

4.6.2 Wide Area Experiment

A wide area experiment was also attempted with the WWW server and proxy server located at the BT Research Laboratories in Martlesham (UK), and YAAT and the control tool at UTS in Sydney (Australia). Two experiments were attempted: one using ISDN between the two sites and another using the Internet. The intent was to demonstrate the difference between using a lossy path via the Internet and a non lossy path across ISDN. The two sites are about 10555 Miles apart. It was expected that the Internet path would be lossy and that the ISDN path would show no loss. The Internet path had 21 hops (determined using traceroute [47]). The Internet path exhibited very little loss: of the order of 1 or 2 percent. By adding redundancy its was possible to remove even this loss at the receiver. The user during playout can select redundancy to overcome the effects of loss. However the impact of this feedback loop is perhaps bizarre. What we are effectively doing is injecting more data into the network in order to overcome the effects of packet loss, which has quite likely been caused by network congestion. Intuitively this is counter to the congestion control policies and mechanisms of the Internet.

The standard PCM audio stream is 64Kbits and the ADPCM stream is 32Kbits. If a 32Kbits ADPCM stream is being sent and loss is detected, we add one level of redundancy creating a 64Kbits ADPCM stream. In this example the packet size is doubled, but the packet rate is not affected. The sample sizes can be varied to change the packet rate. In the ISDN experiment a single 64Kbits channel was used. Due to the overhead of the the IP+UDP+RTP headers, the date rate of a 64Kbits PCM stream exceeds the 64Kbits threshold of the single channel ISDN connection. It was expected that a little loss would be detected in this case. Rather stunningly, between 80-90 percent loss was detected and the audio was not intelligible. Even more strangely, sending a 32Kbits ADPCM stream with 32Kbits ADPCM redundancy exhibited only 1-2 percent loss. We have explored the reasons for this apparently peculiar behaviour.

A packet containing one level of redundancy, for example two ADPCM streams, is slightly larger than a single PCM packet using the same sample size due to the few extra bytes in the header required to support redundancy. An ADPCM stream with one level of redundancy therefore generates a higher data rate than a PCM stream with no redundancy. Both streams are of the the order of 64Kbits, but the PCM stream was showing a loss of 80-90 percent and the ADPCM+ADPCM stream was exhibiting only 1 or 2 percent loss.

The explanation lies in the fact that both sites are using Ascend ISDN routers. Both the routers had STACKER LZS compression enabled at the link level during our experiments. When the compression was disabled the high loss rate dropped to a few percent. It can only be assumed that the compression was increasing the size of the packets or the computation was causing the loss. It seems ironic that a telephone line which is designed to take PCM audio should so severely disrupt PCM audio when it is sent as data, between Ascend routers..

4.6.3 Multicast Reception

Another experiment attempted was to listen to the 37th IETF in San Jose December 9-13 1996, using YAAT. An interesting feature of using YAAT compared to VAT [43] and RAT [38] (the other commonly used audio tools) is that YAAT is designed only for playout, and not for conferencing. Thus it has a larger playout buffer than the other two tools, and is better able to cope with network jitter. The perceived quality of the audio when using YAAT seemed better than using VAT or RAT. The larger playout buffer and, in face of loss, playing the last packet rather than silence, seemed to be the reason for this improved quality. Although RAT also repeats the previous packet in face of loss the larger playout buffer seemed to improve the quality of the audio.

4.7 Related Work

Sun Microsystems have a web server, Jeeves , which allows Java code to be pushed into the server (servlets). The Java servlets run in a restricted environment, due to the security concerns of downloading arbitrary code into a server. Using a Jeeves web server as a replacement for the proxy server is not currently feasible for security reasons.

Our work is significantly different to that embedded in existing WWW-based streaming tools. For example, Progressive Networks have a tool called RealAudio which supports streaming audio. Special purpose Audio servers are used for the playout of the audio. Code cannot be pushed and pulled, and support for adaptation is limited.

“Towards an Active Network Architecture” [83], describes a scenario where perhaps Java or Safe-TCL code is pushed into nodes in the network. This work seems more geared to a lower level in the network infrastructure, and may face serious management problems such as deployment. What we are proposing with our protocol server is an application level, end-to-end solution, which we believe is both efficient and realistically deployable.

Our work is an experimental contribution to research into active networks within a Java framework. Our focus has been on efficiency and deployability, which begins to test the limits of what active functional elements can be reasonably deployed using an infrastructure that is available today.

Our work has some similarities with transcoder work where, at tails of the network, a high bandwidth stream needs to be transformed to a lower bandwidth stream. The protocol server could be used to push protocol transcoders to the intersections of the high/low bandwidth networks.

4.8 Discussions and Conclusions

Another application has been chosen, that of audio streaming and again as in the *slogin* and MMD cases there seems to be utility from using an ALP approach.

We have shown that using the Java programming language it is possible to build a real time audio playout application. On high end workstations interpreted byte codes are fast enough to support such a real time application. Just in Time compilers will be available for most platforms, so performance worries should disappear.

The object oriented nature of Java enables encapsulation and hence simple reuse of various components of the protocol stack such as the RTP code and the various codecs. Java based ALPs are possible and have been shown to work. However the Java programming language makes packet header processing quite complicated and error prone.

We have built a set of codecs and tools to enable the playing out of real time streams. The next stage of the project was to design an API for mobile protocol stacks, and then build a protocol server to validate our ideas. A simple ALP which we intended to build is a simple compressor and decompresser, to demonstrate that the ALP API which is developed is flexible enough to support a totally different style of interaction.

We intended to build a protocol server on which a whole protocol stack will reside, along with codecs. So, ultimately, not only will the protocol stack be pushed to the proxy server but the audio tool will also pull its component of the stack. We envisage a model where whole stacks will be available on protocol servers. A tool such as a WWW browser can perhaps pull a decompresser, and push a compressor to the WWW server. The protocol server is independent of either the server or client application - it is just a repository of possible protocol stacks to use. The protocol server is not limited to being used by just WWW applications. For example, new components of the telnet protocol stack could be pulled to the client and pushed to the server, such as encryption/decryption functions.

The protocol server idea solves a large set of problems. Protocol stack compatibility issues disappear if the whole stack is available in one place. The client and server components will interoperate. New stacks can be deployed rapidly. In an ideal situation only one implementation of a stack may be necessary. For example, if a new audio codec is developed then all audio tools could dynamically incorporate this codec - the ultimate in software reuse.

At this point it looked as if all that was required was to define an API for ALPs and to build protocol servers. Defining an API that captured the semantics of all possible ALP stacks turned out not to be easy. An example of one of the problems is given in the final chapter (7). It also transpired that a special purpose protocol server was not required. A web server is a perfectly good repository for dynamic code as shown in the next chapter (5).

The solution finally adopted for dynamically deploying ALPs is described in the next chapter (5).

Chapter 5

Application Level Active Networking

5.1 Introduction

The work presented in this chapter is joint work with Michael Fry and Glen MacLarty [36].

In this chapter we link together the ideas presented in the *slogin*, MMD and JavaRadio chapters (2, 3 and 4). Thus far we have concluded that benefit can be derived from using ALPs. In the JavaRadio chapter another ALP was developed. It was built using Java so that in the next step the ALP could be downloaded and used by applications. Defining a generic API between an application and a stack is complicated. The complexity is due to trying to capture all the possible application requirements. An ALP may possibly include audio codecs. Therefore all applications in a transaction need to download the ALP in order to communicate. Standard APIs to network stacks provide a small set of calls (e.g. open, accept, close, send, receive). Consider an audio ALP, the send request will take data in one audio format transcode it to another format, frame it in RTP perhaps and then transmit it. The API to an ALP would therefore seem to be analagous to a standard network API. The intent was to define such an API.

One problem that arose was that of RTP redundancy [69]. In this scheme more than one payload is carried in a packet. A packet contains the current audio sample and a previous audio sample. Typically the previous audio sample is encoded with a codec that provides higher compression than that used with the primary sample. We have considered an API that when passed data encodes it with a codec and then transmits it. The question arises how will this simple API support re-encoding the previous sample that it was given? A simple solution may be to pass in two buffers to be encoded as the primary and the secondary samples. The problem becomes more complex when one considers that arbitary levels of redundancy are allowed by the specification. This is one example of a problem that was encountered. If functionality is added to the ALP the application may require upgrading. All we would have achieved is moving the problem around.

Therefore instead of separating the application and the ALP they are merged into a single entity a *proxylet* which is loaded into the network. This structure is termed Application Level Active Networking (ALAN). This chapter describes the concepts involved and and a particular implementation *funnelWeb*.

As the Internet has evolved the time to develop and deploy new protocols and services has been increasing. This increase in deployment time is due to a number of factors. As the Internet has become more commercially important, there is a natural tendency to technical stability that discourages experimentation. Standardisation processes are becoming more lengthy, involving more people. With the increasing number of routers and end systems the deployment time for new protocols is also increasing.

A proposed solution to this deployment blockage is Active Networks [83]. This solution proposes that network packets carry active code which can be executed in routers. This allows for either new protocol deployment or for single active components to be deployed for one-off use. We believe that, while this solution is elegant, issues of performance, security and ownership will prevent a packet based approach from gaining acceptance.

There are clearly performance implications for executing 3rd party code dynamically on routers. The potential impact on the router fast path is likely to be unacceptable. Nonetheless some of these issues are being investigated using FPGAs, but this is not yet applicable to the current network infrastructure.

There are many security issues at various levels. At what level should dynamic code be trusted? Whose code can be run in a foreign router? It is unlikely that an ISP will allow a competing ISP to run code on its routers. The potential for network attacks increases severely if active code can be loaded into routers. In essence: a router is such a critical network component that it is unlikely that those in control of such entities will allow even the simplest pieces of code to be dynamically loaded onto them by a third party. Although the owner of a router may dynamically load code into a router.

The approach we have developed is to deploy active elements as application level entities. Thus we have previously proposed and validated an Application Level Active Networking (ALAN) framework [31]. There are two key components of this approach. Active entities are called *proxylets*. *Proxylets* are written in Java and are identified by reference. A *proxylets* can be downloaded and executed on a special proxy machine called an Execution Environment for *Proxylets* (EEP).

EEPs exist at strategic points of the network in an analogous manner to Web caches. They allow dynamic deployment of value-added services. We envisage that EEPs will be run on end systems as well as inside ISP networks. Many ISPs now allow client Web servers to be placed inside their networks in order to benefit from high bandwidth connectivity. We are starting to see systems placed into ISP sites for content distribution [6] and to aid reliable multicast [44]. We believe that placing EEPs in ISP sites provides a more flexible solution.

The purpose of this chapter is to describe and justify the current state of the ALAN infrastructure. This is associated with a software release. The chapter is organised as follows. We firstly provide an overview of the basic ALAN approach using a Web-based example. We then describe in detail the major components, which are the EEP, the *proxylet*, including permanent system *proxylets* that are used for discovery and routing. We discuss implementation issues and lessons learned from deployment. We provide some performance measures to justify our belief in ALAN.

5.2 ALAN Overview

5.2.1 Basic ALAN

We have previously described an ALAN system[31]. This system is designed to enable the enhancement of communication between regular Internet clients and servers, such as WWW browsers and servers. It also provides a platform for the dynamic deployment of active code elements. This system has been implemented and tested.

Communication is enhanced by one or more Execution Environment for *Proxylets* (EEP) that are located at optimal points of the end-to-end path between the server and the client. (In our first reported version the EEPs were called Dynamic Proxy Servers (DPSs). We have now changed the name of these entities to one that we believe is more descriptive). It is possible to download protocol entities, called *proxylets*, onto the EEP infrastructure. These *proxylets* then act as filters or enhanced protocol func-

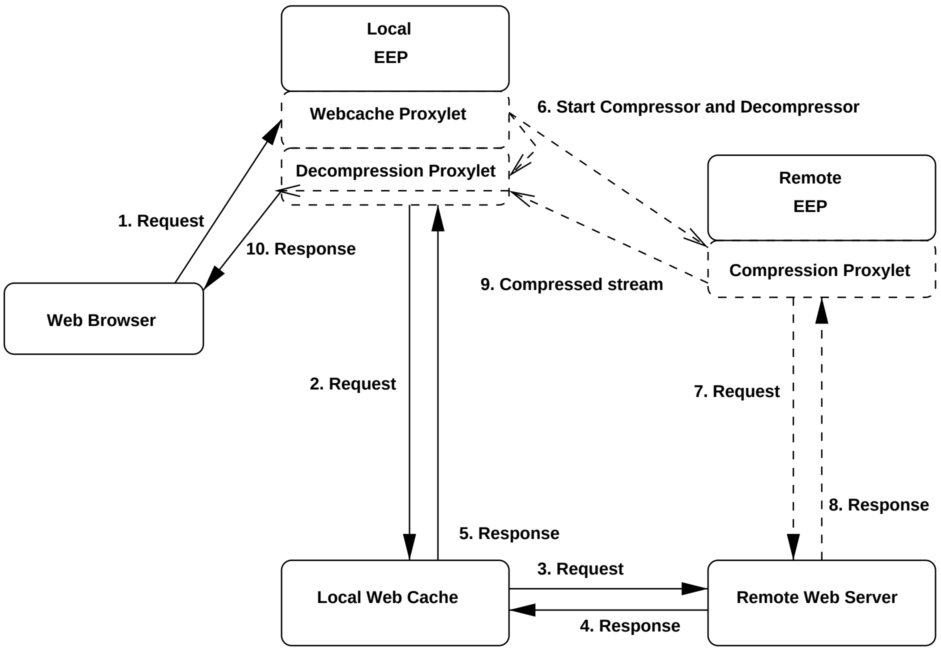


Figure 5.1: Text Compression

tionalities that improve the level of service between servers and clients. *Proxylets* are implemented in Java. They are stored as single Java Archive (JAR) files on WWW servers and hence are referenced and accessible via a URL.

An example scenario is as follows. A large text file resides on a WWW server in the UK. A user in Australia wishes to download this text file. The end-to-end path between client and server typically includes some congested and/or low bandwidth network clouds. This results in high latency page downloads. However, if we are able to compress the text page near the source, transmit the compressed text to the client, where it is decompressed and rendered, this will result in improved download latency. Furthermore, it will also reduce bandwidth costs at the receiver as determined by inbound volume.

One approach to this would be to implement the optimisation at the server and the client. However this is not a generic or scalable solution. Rather, we produce the same outcome in a manner that is transparent to both servers and clients.

The key to this transparency is the use of a Webcache *proxylet*. Figure 5.1 shows the interactions involved. An EEP is run at the site from where the user is browsing. A Webcache *proxylet* is started on the local EEP and the proxy variable in the users browser is set to the Webcache *proxylet* EEP machine. (This is the only non-transparent aspect of the optimisation, however it would seem a reasonable step to take when installing and configuring a browser). The Webcache *proxylet* does not actually perform any caching, but this scheme allows the Webcache *proxylet* to be in the path of HTTP requests.

In the following description the numbers in parentheses are references to the interactions in Figure 5.1. A HTTP request involves a request for a page on a WWW server (1-3). Preceding the requested page in the returned stream is a MIME content type defining the content of the stream (4-5). This allows the browser to correctly render the incoming stream. The content type “text/html”, normally causes the browser to download and render the text.

The Webcache *proxylet* is in the position to observe the content types of all responses being transferred (5). This is done by the Webcache performing a HTTP HEAD request to the server to retrieve the header information such as Content-type and Content-Length. The Webcache *proxylet* has a table of mime content types on which it is able to perform special operations. If a mime content type has no table entry, then the Webcache *proxylet* simply relays the response to the browser. These are content types to which the system can add no value.

In our example, if the Webcache *proxylet* observes that the content type returned is “text/html”, it knows to perform special processing. The Webcache *proxylet* firstly attempts to identify an EEP which is close to the source of the text, i.e. the WWW server on which the text page resides. In the initial version of ALAN there was prior knowledge of the location of appropriate EEPs. However in the current version this location process is automated.

Once an appropriate EEP is identified a text compressor *proxylet* is started by the Webcache *proxylet* on the remote EEP. At the same time a decompresser *proxylet* is started on the local EEP (6). In both cases a URL reference to the appropriate JAR file is given to the EEP so that it can download and start the *proxylet*.

The compression *proxylet* is also told the URL of the text file and the location of the decompresser *proxylet*. Thus the compressor makes a standard HTTP request for the text file (7), then compresses the incoming text (8) which is sent to the decompresser *proxylet* (9). The decompresser then simply decompresses the text and sends it to the web browser (10).

Further implementation details and performance measurements are provided in [31]. This includes further examples, including transcoding of Web-based audio streams, and some non Web-based *proxylets*. The system as described however has some deficiencies.

5.2.2 ALAN Enhancements

In subsequent sections we describe the current status of the two key components: the *proxylet*, the EEP. However we have also further enhanced the infrastructure to address two key shortcomings evident in the system described above.

Firstly, Figure 5.1 shows a *proxylet* providing compression of content. One of the key requirements for this service to be beneficial is for the content to be compressed from a point as close as possible to the source. This implies that we need to locate a cooperating Webcache which is close to the source web server. In our first version the

EEP host to contact is defined within a configuration file. This method is not scalable, and does not encompass the dynamic changes which occur in the network hierarchy. Our current version implements an initial solution to this problem, but ongoing research is investigating a more generic solution.

Essentially, this is an issue of EEP location and information routing. A related issue that is relevant to the scenario described is that it is not always desirable to compress text. The page may be too small to justify compression, and or there will be no significant latency savings due to network conditions. This can be viewed as an application level routing problem, and also one of policy based retrieval.

A further shortcoming with our initial approach is that content that is compressed from a remote *proxylet* will bypass the upstream cache. This results in the object not being stored in the cache, even if the content is identical to the original source object. This is a major shortcoming, as future requests cannot be served from a cache, but will create an additional request to the original server.

Furthermore we may wish to give users some level of control over how content is delivered. For example, an organisation with a high-speed LAN and remote user access via a modem bank, may install a webcache. This would allow remote modem users to require that all data is compressed before transfer across the modem link, decreasing object size and download latency. Users connected directly to the LAN however, may prefer no content processing as this may slow down delivery time across the high-speed link.

We discuss our enhancements to meet the issues articulated above following our detailed description of *proxylets* and EEPs.

5.3 *Proxylets*

A *proxylet* is code which is downloaded and run on an EEP. The name was chosen with regards to applets which run in web browsers, and servlets which run on web servers. The *proxylet* is an entity which runs in the network.

5.3.1 Design

The *proxylet* is designed to be an entity which is *loaded* onto an EEP and then *run*. A *proxylet* is loaded onto an EEP by reference. Using a reference means that the entity which is loading the *proxylet* onto an EEP does not have to have a copy of the *proxylet*. Loading by reference means that only one copy of a *proxylet* needs to exist. This addresses some of the problems that occur with version mismatches, since there is only one copy of a *proxylet* and it is the definitive copy.

It was also decided that there would be no owner of a *proxylet*. There is no permanent handle to a *proxylet*. Once a *proxylet* is loaded and run, anybody with the correct interface can connect to the *proxylet*, stop it or send new parameters. The benefit is that a control channel does not have to be held open to a running *proxylet*. Using a control channel would have had other ramifications in terms of the controlling node failing, or the case where control needs to be handed over. A *proxylet* can have a permanent connection to a controlling entity if required.

A *proxylet* is a fire and forget entity. There will be many cases where a *proxylet* is started by another *proxylet* or program. It is thus not clear where the control should reside. If required an entity that started a *proxylet* may retain an association. However a feature of having no controlling entity is that if an error occurs, there is no obvious place to send the error message.

A mechanism is required to stop a *proxylet*. Also a mechanism is required to send a message to a *proxylet* while it is running.

A *proxylet* does not have access to the network interface at a low level in terms of packet filters. Some active network schemes allow entities to manipulate extant flows. We provide no such special hooks to *proxylets*. In terms of deployment special permissions would be required to allow the EEP to manipulate flows. There would also be no portable way to provide such features. It was also felt that most applications that made use of *proxylets* will be aware that they are using *proxylets*. So there is no requirement to hijack or manipulate flows.

A *proxylet* is not able to use or affect local resources on the host on which it is running. The only exception to this rule is the ability to create network connections. A *proxylet* is not able to create files or look at the contents of files. It is not able to spawn processes on the host on which it is running. This secure feature of a *proxylet* should mean that sites can run EEPs without being concerned about possible security violations (apart from possible denial of service attacks caused by generating large amounts of network traffic).

5.3.2 Implementation

Proxylets are written in Java and have to adhere to a *proxylet* interface. This is shown in figure 5.2. This is similar to the interfaces that applets or servlets have to adhere to.

It would have been possible to provide special mechanisms for *proxylets* to create network connections. Instead it was decided to allow a *proxylet* to just use the APIs available to standard Java programs. We didn't want to limit the scope of possible *proxylets* by reducing the set of network options available to them. Using the standard Java networking APIs makes it relatively simple to create *proxylets* from existing network

```
public interface ProxyletInterface {
        /**
         ** This is the first method to be called in the proxylet.
         ** It is expected that this method will just store its arguments
         ** and return.
         **
         ** @param args The initial arguments. Note that args[0] is the
         ** class name.
         **
         **/
        abstract void init(String args[]) throws Exception;

        /**
         ** This method is called after the init method is called. Once this
         ** method returns the proxylet is terminated.
         **
         **/
        abstract void run() throws Exception;

        /**
         ** This method may be called while the proxylet is running to provide
         ** new arguments.
         ** @param con New parameters to the proxylet.
         **/
        abstract void control(String con) throws Exception;

        /**
         ** This method is called to notify the proxylet to stop. If the
         ** proxylet does not stop it will be killed in a number of seconds.
         **
         **/
        abstract void stop() throws Exception;
}
```

Figure 5.2: Interface implemented by a *proxylet*

code.

We also considered providing special channels for simple inter-proxylet communications, as it was clear that *proxylets* would be required to communicate with each other. It turned out that it was simpler to just allow *proxylets* to use whatever mechanism was convenient. The most convenient method used by *proxylets* has been RMI. A nice side effect of *proxylets* using RMI for inter-proxylet communication is that standard programs can make use of services provided by *proxylets*. A service may be made up of a number of *proxylets*, such as the Webcache *proxylet* (Section 5.2.1). A *proxylet* may for example need to communicate with another *proxylet* in order to discover a service.

A standard way of bundling together a set of Java classes is to use a Java archive JAR file. Physically a *proxylet* is just a JAR file. Initially we had considered writing special servers on which *proxylets* would reside. It was however decided to just refer to *proxylets* via Universal Resource Locators URLs. It therefore followed that *proxylets* could reside on WWW servers. This removed the requirement for having special servers.

5.3.3 Lessons learned from deployment

A number of issues have arisen during deployment. One of the issues is to do with security. As stated above a *proxylet* once running can potentially be controlled by anybody with the correct interface. This unfortunately meant that anybody could also kill a running *proxylet*. In fact it was observed that the graphical interface to running *proxylets* made it rather too easy to kill *proxylets*.

As a temporary security measure a scheme was designed that only allowed *proxylets* to be manipulated from a nominated host. A mechanism already existed to allow what java calls “properties” to be passed through to *proxylets*. A file called “METADATA” containing properties can be placed in the jar archive that is a *proxylet*. The properties in the METADATA file can then be made available to a *proxylet*. This scheme can be used to pass in configuration information to a *proxylet*. Some new properties were created which, if present, state from which host connections to a *proxylet* are allowed.

Some of the *proxylets* that have been written need to contact other *proxylets* running on the same EEP. Currently a property “java.rmi.server.hostname” is used to determine the name of the host. This information is required so often that the *proxylet* stub should return this information.

5.3.4 Future directions

A *proxylet* is currently housed in a simple jar file. One of our partners in the ALPINE project, Lancaster University, has implemented *proxylets* as XML files. Using an XML file allows for conditional statements to be placed in the file to determine what class files should be loaded. This has been used mainly to support the notion of architecture specific code. The architecture of the machine is determined and, if available, native code can be loaded. However as soon as it is possible to load native code, the notion of safe code is jeopardised.

The current way of parameterising a *proxylet* is to either pass it an argument or to set a property in the METADATA file. To see what is in a METADATA file it is necessary to extract the file from the jar file that contains it. However if a *proxylet* is an XML file, then it would be simple to see what the properties are and to create new *proxylets* with different properties. It would also be possible to place conditional code in the XML file, such as alternative locations of the jar file that contains the class code.

More complex security settings could also be placed in the XML file.

5.4 Execution Environment for *Proxylets* - EEPs

5.4.1 Design

An EEP is the entity on which a *proxylet* is run. It has two interfaces. The control interface is used to load, run, modify and stop *proxylets*. The monitor interface is used to monitor an EEP.

The distinction between the two interfaces is intended to denote the difference in use. The control interface is used by a client to manipulate a particular *proxylet*. The control interface consists of a number of methods:

- Load

  This method is used to load a *proxylet* onto an EEP. A URL is passed to this method.

- Run

  Once a *proxylet* is loaded it is started with the arguments to this method.

- Modify

  Once a *proxylet* is running it can be passed new arguments by this method. The arguments are passed into the *proxylet* via the control method that it must provide.

- Stop

  This is a method to stop a *proxylet*. Calling the stop method, causes the stop method in the *proxylet* to be called. A *proxylet* is therefore provided with an opportunity to cleanly terminate. If a *proxylet* ignores the stop method it will be killed anyway after a number of seconds.

The control methods are mapped onto the methods that a *proxylet* must provide, as can be seen in Figure 5.2.

The monitor interface is intended to be used by the owner of an EEP to monitor activity on an EEP. It is rather like a process monitor, such as the “ps” or “top” commands found on UNIX systems. The “top” command typically polls the system at one second intervals and then displays a list of process. Rather than use a polling scheme, which is wasteful of network bandwidth, a call back scheme is used. A registration is made with an EEP, and when there is a change of state the EEP sends a message to all registered parties.

The methods of the monitor interface are as follows.

- register

  This is how a monitoring entity registers interest in an EEP. A handle is passed to the EEP to denote where the callback should go. If the EEP is retrieving a *proxylet* through a cache, then the host and port of the cache are also returned.

- unregister

  This is used to detach a monitoring entity.

- version

  This returns the version number of the EEP.

- *proxylet*

This returns all that is known about this *proxylet*, such as where it was loaded from and with what arguments it was started.

5.4.2 Implementation Issues

The EEP is written in Java, as are the *proxylets*. The current implementation uses a separate Java VM for each *proxylet*. This is wasteful in terms of resources, but does make it relatively easy to manage *proxylets*. A *Proxylet* can easily be killed by killing the VM in which it is running. Using a separate VM also means that *proxylets* cannot interfere with each other.

A *proxylet* is actually run by downloading the jar file to a local file system and then using the standard Java classloader to run the *proxylet*. Using the normal classloader allows the standard Java security permissions file to be used to restrict the permissions of *proxylets*.

The *proxylet* location is actually added to the “java.rmi.server.codebase” property. This has to be done to allow *proxylets* to register services with the “rmiregistry”. The “rmiregistry” has to be able to access the code that is registered with it. A typical way of using RMI is that a piece of code is started with the “codebase” property pointing at the location of the code. When a call is made to the “rmiregistry” to make a class available across the network, “codebase” is used to locate the code. An unfortunate limitation of this scheme is that only one code location can be set.

As has already been mentioned, there is no obvious place where errors caused by *proxylets* should be sent. No special mechanism is provided to *proxylets* to register errors. In our initial experiments a *proxylet* writer would typically have access to the EEPs. Errors generated by *proxylets* would appear in the window from which the EEP was run.

Certainly on Unix systems there are two streams on which output can be generated: a standard output stream and a standard error stream. Java provides calls to send output to these two streams. Rather than invent a new way of generating output it was decided to continue with this way of generating output.

These two streams are captured by the EEP from the *proxylet*, and then multicast at a site-wide time to live (TTL). Any output generated by legacy code will also be correctly captured. Simply multicasting the error at a site-wide TTL means that no per site configuration is necessary. A simple logger can log all messages generated by all *proxylets* at that site. The format of the message is such that the *proxylet* and stream on which the message was generated can easily be identified. A simple program is provided to print out the error messages generated at a site. However the issue of errors generated at a foreign site is not currently addressed. Our plans in this regard are discussed below.

Currently the EEP provides very little to aid a *proxylet*. A *proxylet* has full access to all of the JDK apart from the components which are protected by the security manager.

An attempt has been made to keep the EEP as small as possible. Only functionality which is absolutely necessary has been placed in the EEP. As will be seen below some of what could be considered to be EEP functionality has been implemented in the form of permanent *system proxylets*.

5.4.3 Lessons learned from deployment

There are a number of obvious deficiencies.

Errors generated by *proxylets* at foreign sites are not handled. However there is support for the user to write a *proxylet* which returns errors from foreign sites.

In the initial design it was felt that only a monitoring agent would need to know what *proxylets* are running on an EEP. However there are cases where a *proxylet* needs to know what *proxylets* are running on a particular EEP. An example may be a *proxylet* which is used to start another *proxylet* on every EEP. In this case the *proxylet* will need to be able to check if the other *proxylet* is already running.

Many *proxylets* need to interact with services running on the local EEP. The EEP should provide the localhost name through the *proxylet* stub.

5.5 Controlling *Proxylets* - The graphical user interfaces

There are currently two separate GUIs associated with *funnelWeb*. One is a control interface. The other is a monitor interface. The separate GUIs are partitioned consistent with the interfaces offered by the EEP.

The control interface is used to load, start, modify and stop a *proxylet*. If a *proxylet* exits for whatever reason this information is not propagated to the control interface, since the control interface performs atomic transactions with a *proxylet*.

The monitor interface shows what *proxylets* are running on an EEP and their current state. This interface can be used to spawn control interfaces which can be used to manipulate individual *proxylets*.

A simple command line interface is also provided to print out all the error messages generated by the EEPs at a site. Typically, when writing and debugging *proxylets* it is necessary to run all three interfaces.

An implementation reason to keep the monitoring and control interfaces separate is that the monitoring interface uses callbacks. The callbacks are implemented by an EEP making RMI calls back to the monitor interface. This causes two potential problems if the interfaces were integrated. Firstly, it would not be possible to start a *proxylet* that is not running a rmiregistry for whatever reasons. The second reason is that it would be difficult to start a *proxylet* on an EEP at a foreign site from behind a firewall. A typical configuration for a firewall is that outgoing connections are allowed and incoming ones are not. Similarly it would not be possible to start *proxylets* from sites that are behind NATs (Network Address Translators).

In the longer term however there is an intent to provide an integrated interface. This can perhaps be achieved by disabling monitoring features, or by implementing callbacks via an alternative mechanism to RMI.

5.6 *System Proxylets*

As has previously been observed, our initial release contained several deficiencies, perhaps the most significant being in the area of dynamic discovery of EEPs and routing. It could be argued that such functions are core to the EEP infrastructure, and thus should be implemented within the EEP. However in practical terms it is more attractive to develop such functionality incrementally as *proxylets*. This allows us to upgrade the functionality of running EEPs relatively easily. In this section we describe two such *proxylets* which represent quite simple, initial approaches to providing routing and error handling functions.

We have developed a simple mechanism for certain *proxylets* to be started at boot time of an EEP. We call these *system proxylets*. At this time *system proxylets* do not have any special privileges. They are also not handled differently to standard *proxylets*.

Their only distinction is that they are loaded at boot time. It should be noted that if either of the two system *proxylets* is not present, the EEP will continue to function, although with slightly reduced functionality.

5.6.1 Routing

The scope of the problem of building a scalable application level routing infrastructure is huge. In order to permit experimentation, we have provided a routing *proxylet* which does not scale and provides a very naive proximity interface. Below we outline our current work towards providing a scalable routing infrastructure [33]. The next chapter (6) goes into greater detail.

The initial routing *proxylet* provides two interfaces. The first interface provides a list of all executing EEPs. The second interface provides the location of a *proxylet* close to a provided domain name.

The implementation of the routing *proxylet* is simple. A coordinator node is preconfigured into the routing *proxylet*. Each *proxylet* periodically sends a registration packet to the coordinator. All EEPs are therefore known by the coordinator.

Any routing *proxylet* can be interrogated for the list of all EEPs. The client does not need to know which routing *proxylet* is the coordinator. If a request is made of an EEP which is not the coordinator then it will request the information from the coordinator and pass it on to the client.

As well as a list of all available EEPs the routing *proxylet* also provides a proximity function. The proximity function has the name “close”. A domain name can be passed to the “close” method of any routing *proxylet* and it will attempt to return an EEP which is close to this domain name. Our implementation is extremely naive: a longest match on the domain name string is performed.

We realize that there is a large number of obvious failure modes with such a scheme. For example, if a request is made for an EEP close to “acm.org” there is no useful geographic information. Another problem is that nowadays a site with, for example, the suffix “.au” does not necessarily have to be in Australia.

Therefore we have provided a routing *proxylet* with many limitations. However it has allowed more interesting *proxylets* to be built. For example, the caching *proxylet* [52] is currently being modified to use the routing *proxylet*. This allows it to locate an EEP close the the source web server.

The routing *proxylet* has an automatic upgrade mechanism. When a routing *proxylet* performs its periodic registration with the coordinator a version number comparison takes place. If the coordinator is running a newer version of the routing *proxylet* than the *proxylet* registering, then the registering *proxylet* loads the latest version of the *proxylet*. This automatic reload mechanism has allowed us to upgrade the implementation of the routing *proxylet*. It automatically deploys across all running EEPs. As we develop more scalable solutions they will be trivial to deploy.

5.6.2 Error Handling

As stated earlier a *proxylet* is a fire and forget entity. Therefore if debugging or error output is generated from a *proxylet* there is no obvious place to send the output.

Currently any output generated by a *proxylet* is multicast at a site wide scope. An application is provided to print out the error messages. This allows “local” *proxylets* to be debugged. However currently there is no way to retrieve error messages from remote sites. A system *proxylet* has been provided for error handling which currently does nothing. Eventually we intend to use it to return error messages from foreign sites. As with the routing *proxylet*, this will be deployed automatically as it becomes available.

5.7 Performance Measurements

We have undertaken some initial measurements to indicate the overhead incurred by the architecture. These results also indicate where performance can be optimised. Measurements were taken using the Webcache example described previously. Overheads can be decomposed into a number of components as follows.

- EEP Location Overhead
- *Proxylet* Load Time
- *Proxylet* Execution Overhead

5.7.1 EEP Location

For the text compression example, the overhead incurred is the time to locate a suitable remote EEP, the time for the remote EEP to download, load and start executing the *proxylet*, and the time that the *proxylet* itself takes to download the requested page, compress it, and transfer it to the local EEP (or Webcache).

For *proxylets* invoked from the Webcache, the decision of which EEP to contact can be made by a call to the local EEPs Routing *Proxylet*. This call incurs a delay and is shown to be around 130 milliseconds. This delay is the overhead of the RMI call and the processing time of the Routing *Proxylet* to determine a suitable remote EEP.

5.7.2 *Proxylet* Load Time

The *proxylet* load time consists of the time taken to download the jar file from the *proxylet* web server, the time to start a new Java Virtual Machine (VM) and the time to synchronise the new VM with the current VM.

Since a *proxylet* is a web object, its code can be cached. Thus the delay to download the *proxylet* code is likely to be small when a particular *proxylet* is loaded frequently.

The size of the *proxylet* bytecode is also relatively small, with most *proxylets* currently being under 6Kb in size. A number of load times observed on a test machine showed a delay of approximately 3000 milliseconds for loading a *proxylet* of around 4Kb from a web server on the local area network. This time consisted of approximately 1000 ms to start the new VM, under 200 ms to download the byte code, with the remainder made up of RMI overhead, synchronisation, security checking, and logging.

These timings will obviously vary for different EEPs, with the hardware performance of the host EEP machine being critical to the speed of starting a VM. The network latency in retrieving *proxylet* code is also a factor which will increase overall load time.

5.7.3 Execution Time

The overhead incurred by the transcoding *proxylet* is dependent on a number of factors.

- Retrieval time for the web object
- Size of request object (for *proxylets* such as the text compressor)
- Overhead of the *proxylet* process
- Speed of the machine on which the *proxylet* is being executed
- Network delay incurred in transmission of transcoded data to client or local EEP

Given that an EEP is selected based on its likely “closeness” to the web object, the retrieval time of the web object will often be small. Again this is dependent on caching and transient network performance.

The remainder of the overhead is dependent on transient network performance, the performance of the *proxylet* transcoding process and the machine on which it is being executed. These overheads will impact the overall efficiency of the architecture, but can be optimised through *proxylet* implementation and location of EEPs that have adequate processing resources.

Cross platform tests have shown the following results for the compression *proxylet*. The compressor was run standalone and an average taken over 10 runs. The web object was a text page containing the full King James Bible comprising 5073934 bytes.

Operating System	Processor	Memory (Mb)	JDK	Time (ms)
Solaris 8	440 MHz UltraSparc IIi	256	1.3	5206
Solaris 8	167 MHz UltraSparc	128	1.3	12814
Linux 2.2.16	733 MHz Intel Pentium III	256	1.3	2219
Windows 2000	733 Mhz Intel Pentium III	256	1.3	2198

Table 5.1: Compression Times

For text compression the benefit of using this architecture is demonstrated when available bandwidth is low and the size of the requested object is large.

A bottleneck however will be the compression rate of the stream at the remote EEP. Using the compression timings from Table 5.1, for the Linux operating system, we can see that the compressor transcodes the King James Bible of size 5073934 bytes in 2219 milliseconds. This extrapolates to a compression throughput of 2286585 bytes per second, or around 18 Mbps.

Therefore we predict, for this implementation, compression is a benefit when the available end-to-end bandwidth is less than 18Mbps.

To support our hypothesis we performed a test of the implementation by downloading the King James Bible from a web server at University College London to a client at the University of Technology, Sydney. This was performed once through our architecture and once using no intermediate caching. It resulted in a download time through our architecture of 4 minutes 23 seconds in comparison to the non-caching download time of 19 minutes 12 seconds.

5.8 Charging

For *funnelWeb* to be the most useful, global deployment would be required, with EEPs running at sites all over the world, and in some cases running in the core of the network. An application layer multicast *proxylet* would be best situated in the core of the network, as opposed to the edges.

The problem is what incentive would there be to allow a third party to run a *proxylet* on your EEP? The obvious and standard incentive is charging to allow a third party to run a *proxylet* on an EEP.

For some *proxylets* making an arrangement to run a *proxylet* on a specific EEP may be reasonable. In the general case the power of *funnelWeb* would be the ability for the EEP to be dynamically chosen. In this case the owner of the EEP and the *proxylet* user would have no prior arrangement. In such a circumstance electronic cash could be passed to an EEP before it allows a *proxylet* to execute.

Resources that are used by a *proxylet* such as CPU cycles or network bandwidth could be paid for with electronic cash. The metering and allocating of resources would be simple to implement.

The key to global deployment of EEPs and the use of *proxylets* may be for owners of EEPs to charge for usage.

5.9 Related Work

Active Networks has been an important area of research since the seminal paper by Tennenhouse et al.[83] We have discussed earlier how our work differs from router level active network research. Our research is probably more close to that carried out in active services[7], but is more dynamically deployable.

Our work bears some similarities with mobile agent technology [50]. An example of mobile agent technology is IBM's aglets[49]. Although mobile agents and *proxylets* may for some functions be interchangeable, there is a difference in motivation. Both *proxylets* and mobile agents can be optimisations that attempt to reduce network load and overcome network latency. *Proxylets,* in the general case, aid in this process by attempting to transform the communication stream and/or perform some intermediate protocol processing. A mobile agent in contrast is typically executed close to a target site, in order to perform an interaction on the user's behalf.

The incentive for much mobile agent work is distributed computing. The mobile agent infrastructure insulates the user from the network. An example is a mobile agent infrastructure allowing an agent to move from one execution environment to another while preserving its state. *Proxylets* could be written to behave as mobile agents, but each *proxylet* would have to written to manage it's own state when moved.

Mobile agent technology seems to rely on knowing *a priori* where the agents should be executed. We in contrast aim to have *proxylets* executed at an appropriate location, based on network metrics. The criteria for placing a mobile agent is computing resource, for a *proxylet* it is networking resource. Hence our interest in Application

Level Routing (ALR).

The Active Cache work of Pei Cao[16] et al implements a migration of code via "cache applets". These are associated with web server objects, and are migrated to local proxies. Cache applets perform optimisations such as improving the caching behaviour of dynamic documents. In our work such optimisation are transparent to the web server.

One primary objective in our ALR work has been a requirement for minimum configuration. Our ALR solution can be considered to be analogous to the self organising work of [48, 15, 93].

Much of the work to date in topology discovery and routing in active service systems has been preliminary. We have tried to draw on some of the more recent results from the traditional (non active) approaches to these two sub-problems, and to this end, we have taken a close look at work by Francis[29], as well as the nimrod project[19].

5.10 Deployment

A release was made of *funnelWeb* 2.0.1 on 21st March 2000.

The release is currently only available to project partners. The release requires minimal configuration to start an EEP. Simple configuration information is required such as the location of the Java release.

As it is expected that the EEP infrastructure should be running permanently, a "cronjob" is used to check that an EEP is running. The cronjob runs every hour. It attempts to connect to the local EEP. If the connection succeeds all is well. If for some reason the EEP is not running, the connection attempt fails and a new EEP is started.

5.11 *Proxylets*

Over time a large number of *proxylets* have been written.

1. TCPbridge

   A simple bridging *proxylet* which relays TCP connections. A detailed explanation can be found in chapter 6 section 6.2.2.

2. Webcache

   A Webcache *proxylet* which can start transcoding and compression *proxylets*. The *proxylet* used in the webcache examples in this chapter.

3. rtptranscoder

   A *proxylet* which can transcode simple audio samples to RTP streams. This *proxylet* is currently used in conjunction with the Webcache *proxylet*.

4. refl

   A simple reflector *proxylet* which has been used to join multicast sessions on foreign sites.

5. WAP

   A WAP *proxylet* which converts HTML to WML.

6. NTP

   A NTP *proxylet*. The NTP *proxylet* is used to determine the roundtrip time between EEPs. Potentially very useful for application layer routing.

7. routing

   Routing *proxylet*.

8. RLC

   Receiver-driven Layer Congestion Control *proxylet* [86].

5.12 Conclusions

The architecture and implementation of the ALAN infrastructure in the form of *funnelWeb* has been described. It is concluded that it is a feasible platform. A number of *proxylets* have been written and the number is growing. Consider the work in earlier chapters on *slogin* and MMD. It would now be possible to implement these application layer protocols as *proxylets*.

In order to remotely log into site A from site B a *slogin proxylet* could be run at each site. A local connection using SSH [94] over TCP could be made to the *slogin proxylet*. The *slogin proxylet* would then send packets to the *slogin proxylet* at site B using the *slogin* protocol. At site B the *slogin proxylet* would create a TCP connection to the SSH daemon. The usefulness of the *slogin proxylet* in this instance would be its low delay characteristics, not for its encryption. If the *slogin proxylet* was being used for security purposes the EEPs and *proxylets* would have to be trusted. The EEPs could only be trusted if the owners can be trusted. In the case of a company allowing remote logins for its employees this would be the case. Assuming that both EEPs were controlled by the company. The *slogin proxylet* could be signed to protect against a Trojan.

The MMD protocol could also be implemented as a *proxylet*. This would certainly aid with deployment. In fact the MMD strategy would work best if the main multicasting component could be placed in the core of the network. If native multicasting is not available using *proxylets* the scheme could fall back to using application layer multicast.

It has been shown that application layer protocols provide advantages, but they are harder to implement than protocols using for example, TCP. Extra work is required in designing and implementing these protocols. The extra effort required to design and implement these protocols, could be justified if it is only done once. This is the solution being suggested here. Only one implementation of *slogin* or MMD needs to exist if they are written as *proxylets*. Every user of the *slogin* or MMD could reference the same *proxylet*. Enhancements or bug fixes could be deployed instantaneously. Total catastrophe would also be easy to achieve. A critical bug in an upgrade would potentially cause every user to use the broken *proxylet*. The point remains that *proxylets* allow rapid global deployment. In the next chapter this feature is used to trivially deploy new *proxylets*.

A component which is missing is routing. In subsection 5.6.1 a very naive routing solution is described. This solution was developed solely to allow the ALAN concept to be tested. In the next chapter (6), a scalable routing infrastructure is described. This is the last piece in the puzzle.

Chapter 6

An Architecture for Application Layer Routing

6.1 Introduction

The work presented in this chapter is joint work with Michael Fry and Jon Crowcroft [33].

In the previous chapter we have proposed, implemented and demonstrated an Application Layer Active Network (ALAN) infrastructure. This infrastructure permits the dynamic deployment of active services in the network, but at the application level rather than the router level. Thus the advantages of active networking are realised, without the disadvantages of router level implementation. However we have previously left unsolved the issue of appropriate placement of ALAN supported services. This is an Application Layer Routing problem. In this chapter we define this problem and show that, in contrast to IP, it is a multi-metric problem. We then propose an architecture that helps conceptualise the problem and build solutions. We propose detailed approaches to the active node discovery and state maintenance aspects of Application Layer Routing (ALR).

Our approach has been validated by developments in the commercial Internet environment. It is the case that some Internet Service Providers (ISPs) will support servers at their sites supplied by third parties, to run code of their (3rd parties') choice. Examples include repair heads [44], which are servers in the network which help with reliable multicast in a number of ways. In the reliable multicast scenario an entity in the network can perform ACK aggregation and retransmission. Another example is Fast Forward Networks Broadcast Overlay Architecture [40]. In this scenario there are media bridges in the network. These are used in combination with RealAudio [64] or other multimedia streams to provide an application layer multicast overlay network.

We believe that rather than placing "boxes" in the network to perform specific tasks, we should place generic boxes in the network that enable the dynamic execution of application level services. We have proposed a ALAN environment based on Dynamic Proxy Servers (DPS). In our latest release of this system we rename the application layer active nodes of the network Execution Environments for *Proxylets* (EEPs). By deploying active elements known as *proxylets* on EEPs we have been able to enhance the performance of network applications.

In our initial work we have statically configured EEPs. This has not addressed the issue of appropriate location of application layer services. This is essentially an Application Layer Routing (ALR) problem.

For large scale deployment of EEPs it will be necessary to have EEPs dynamically join a mesh of EEPs, with little to no configuration. As well as EEPs dynamically discovering each, other applications that want to discover and use EEPs should also be able to choose appropriate EEPs as a function of one or more form of routing metric.

Thus the ALR problem resolves to an issue of overlaid, multi-metric routing.

This chapter is organised as follows. We first describe our ALAN infrastructure by way of some application examples. These examples reveal the Application Layer Routing issues that require solution. We then propose an architecture that aids conceptualisation and provides a framework for an implementable solution. This chapter concentrates on the issues of node (EEP) discovery and state maintenance.

6.2 Application Layer Active Networking

The previous chapter 5 described the ALAN architecture and its rendition in the form on the *funnelWeb* [34] package.

6.2.1 WWW cache *proxylet*

We have written a number of *proxylets* to test our ideas. Possibly the most complicated example has been a webcache *proxylet* [53].

The webcache *proxylet* is described in the previous chapter 5. Fig. 6.1 is repeated here to aid the description.

A limitation of our initial experiments is that the locations of the various EEPs are known a priori by the webcache *proxylet*. Another problem is that it is not always clear that it is useful to perform any level of compression. For example if a WWW server is on the same network as the browser it may make no sense to attempt to compress transactions.

From this application of *proxylets* two clear requirements emerge which need to be satisfied by our application layer routing infrastructure. The first requirement is to return information regarding proximity. A question asked of the routing infrastructure may be of the form: return the location of an EEP which is close to a given IP address.

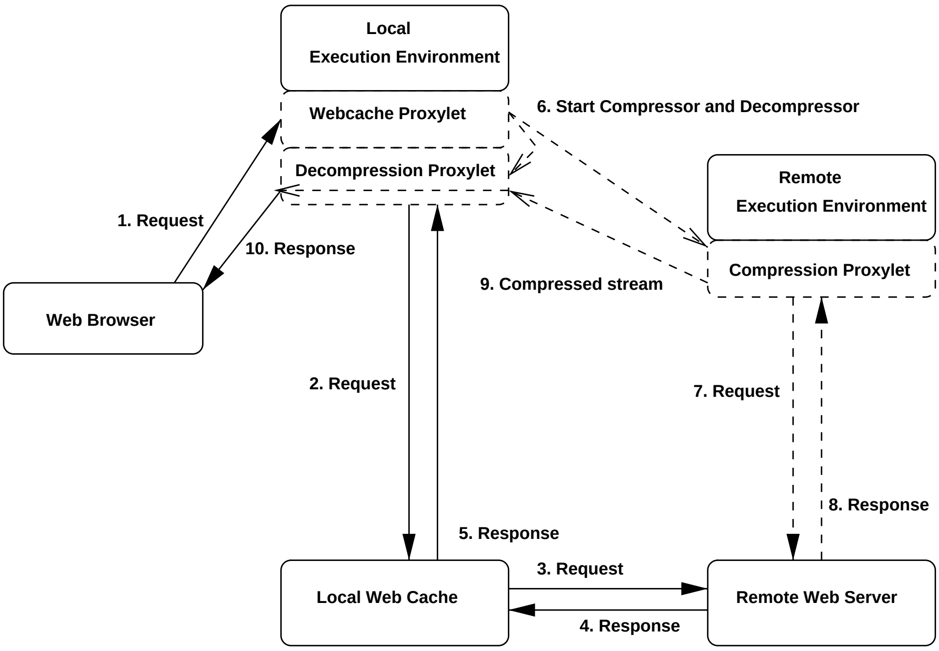


Figure 6.1: Text Compression

Another requirement could be the available bandwidth between EEPs, as well as perhaps the bandwidth between an EEP and a given IP address. An implied requirement also emerges. It should not take more network resources or time to perform ALR than to perform the native transaction.

Given this location and bandwidth information the webcache *proxylet* could now decide if there were any benefit to be derived by transcoding or compressing a transaction. So a question asked of the application layer routing infrastructure may be of the form: find a EEP “close” to this network and also return the available bandwidth between that EEP and here.

6.2.2 TCPbridge

One of our simplest *proxylets* is a TCPbridge *proxylet*. The TCPbridge *proxylet* runs on an EEP and accepts connections on a port that is specified when the *proxylet* is started. As soon as a connection is accepted a connection is made to another host and port. This *proxylet* allows application layer routing of TCP streams. It could obviously be extended to route specific UDP streams.

We have experienced the benefits of a TCPbridge by using telnet to remote log in to computers across the globe. A direct telnet from one computer to another across the global Internet may often experience very poor response times. However if one can chain a number of TCP connections (essentially, source routing) by logging into intermediate sites, better performance is typically achieved. This is because the segmented TCP connections respond more promptly to loss, and do not necessarily incur the overhead of end-to-end error control.

A requirement that emerges from this scenario is the need for the application routing infrastructure to return a number of EEPs on a particular path. We may ask of the ALR: give me a path between node A and node B on the network which minimises delay. We may also ask for a path between node A and B which maximises throughput, by being more responsive to errors. It may also be possible that we require more than one path between node A and node B for fault tolerance.

6.2.3 VOIP gateway

A *proxylet* that we intend to write is co-located with a gateway from the Internet to the PSTN. The idea is that a person is using their PDA (Personal Digital Assistant) with a wireless network interface such as IEEE 802.11 or perhaps Bluetooth [11].

Using a packet audio application such as “vat”, the user wishes to make a voice call to a telephone via an IP-to-telephony gateway. The simple thing to do would be to discover a local gateway. A more interesting solution would be to find the closest gateway to the telephony endpoint. An argument for doing this might be that it is cheaper to perform the long haul part of the connection over the Internet rather than by using the PSTN.

This adds another requirement: the ability to discover information about available services. The normal model for service discovery is to find a service in the local domain. We have a requirement for service discovery across the whole domain in which EEPs are running. So a *proxylet* on an EEP which is providing a VOIP gateway may want to inject information into the routing infrastructure which can be used by VOIP aware applications.

6.2.4 Multicast

With the seeming failure of wide area multicast deployment it starts to make sense to use *proxylets* inside the network to perform fanout of streams, as well as perhaps transcoding and retransmission. A requirement that emerges for multicast is that there is enough information in the routing infrastructure for the optimal placement of fanout points.

6.3 Application Layer Routing Architecture

A number of routing requirements for our ALAN infrastructure have emerged from the examples described in the previous section. In essence our broad goal is to allow clients to choose an EEP or set of EEPs on which to run *proxylets* based on one or more cost functions. Typical metrics will include:

- Available network bandwidth.
- Current delay.
- EEP resources.
- Topological proximity.

We may also want to add other constraints such as user preferences, policy, pricing, etc. In this chapter we focus initially on metric-based routing.

An Application Layer Routing (ALR) solution must scale to a large, global EEP routing mesh. It must permit EEPs to discover other EEPs, and to maintain a notion of "distance" between each EEP in a dynamic and scalable manner. It will allow clients to launch *proxylets* (or "services") based on one or more metric specifications and possibly other resource contingencies. Once these services are launched, they become the entities that perform the actual "routing" of information streams.

We therefore propose an ALR architecture that has four components. In this section we simply provide an overview of the architecture. The four components are as follows.

1. EEP Discovery.

2. Routing Exchanges.

3. Service Creation.

4. Information Routing.

EEP discovery is the process whereby an EEP discovers (or is able to discover) the existence of all other EEPs in the global mesh. In our current implementation (described below) all EEPs register at a single point in the network. This solution is clearly not scalable, requiring the introduction of a notion of hierarchy. Our proposed approach is described in the next section. The approach addresses both the arrival of new EEPs and the termination or failure of existing EEPs.

Routing exchanges are the processes whereby EEPs learn the current state of the ALAN infrastructure with regard to the various metrics. On this basis EEP Routing

Tables are built and maintained. The routing meshes embedded in routing tables may also implement notions of clusters and hierarchy. However these structures will be dynamic, depending on the state of the network, with different structures for different metrics. The state information exchanges from which routing information is derived may be explicitly transmitted between EEPs, or may be inferred from the observation of information streams.

Service creation is the process whereby a *proxylet* or set of *proxylets* are deployed and executed on one or more EEP. The client of this service creation service specifies the *proxylets* to be launched and the choice(s) of EEP specified via metrics. The client may also specify certain service dependencies such as EEP resource requirements.

The service *proxylets* to be launched depend entirely on the service being provided. They encompass all the examples described in the previous section. For example, the webcache *proxylet* launches transcoders or compressors according to mime content type. The metric used here will be some proximity constraint (e.g. delay) to the data source, and/or available bandwidth on the path. The TCPbridge *proxylets* will be launched to optimise responsiveness to loss and maximise throughput. The VOIP gateway *proxylet* will require a telephony gateway resource at the EEP.

Information routing is the task performed by the *proxylets* once launched. Again the function performed by these *proxylets* are dependent on the service. It may entail information transcoding, compression, TCP bridging or multicast splitting. In each case information is forwarded to the next point(s) in an application level path.

The rest of this chapter is devoted to describing our more detailed proposals for EEP Discovery and Routing Exchanges.

6.4 Discovery

6.4.1 Discovery phase

We will now describe how the discovery phase takes place. The discovery phase is implemented by a "discovery *proxylet*" that is pre-configured with each EEP. Since *proxylets* are loaded on EEPs by URL reference, it is trivial to update the version of the discovery *proxylet*- it will be automatically loaded when a EEP starts up.

The function of the discovery phase is for all EEPs to join a global "database", which can be used/interrogated by a "routing *proxylet*" (discussed further in the next section) to find the location of an EEP(s) which satisfies the appropriate constraints. Constructing this database through the discovery phase is the first stage towards building a global routing infrastructure.

Requirements

There are a number of requirements for our discovery phase:

1. The solution should be self configuring. There should be no static configuration such as tunnels between EEPs.

2. The solution should be fault tolerant.

3. The solution should scale to hundreds or perhaps thousands of deployed EEPs.

4. No reliance on technologies such as IP multicast.

5. The solution should be flexible.

The discovery protocol

The solution involves building a large distributed database of all nodes. A naive registration/discovery model might have all registrations always going to one known location. However it is obvious that such a solution would not scale beyond a handful of nodes. It would not be suitable for a global mesh of hundreds or thousands of nodes.

In order to spread the load we have opted for a model where there is a hierarchy. Initially registrations may go to the root EEP. But a new list of EEPs to register with is returned by the EEP. So a hierarchy is built up. An EEP has knowledge of any EEPs which have registered with it as well as a pointer to the EEP above it in the hierarchy. So the information regarding all the EEPs is distributed as well as distributing where the registration messages go. The time to send the next registration message is also included in the protocol. So as the number of EEPs grows or as the system stabilises the frequency of the messages can be decreased.

If an EEP that is being registered with fails, the EEP registering with it will just try the next EEP in its list until it gets back to the root EEP.

With this hierarchal model, if the list of all EEPs is required then an application (normally this will be only routing *proxylets*), can contact any EEP and send it a node request message. In response to a node request message three chunks of information will be returned: a pointer up the hierarchy where this EEP last registered; a list of EEPs that have registered with this EEP if any; a list of the backup EEPs that this EEP might register with. Using the node message interface it is possible to walk the whole hierarchy. So either an application extracts the whole table and starts routing *proxylets* on all nodes, or a routing *proxylet* is injected into the hierarchy which replicates itself using the node message.

The discovery *proxylet* offers another service. It is possible to register with the discovery *proxylet* to discover state changes. The state changes that are of interest are a change in where registration messages are going, a EEP which has failed to re-register being timed out, and a new EEP joining the hierarchy.

In the discussion above we have not said anything about how the hierarchy is actually constructed. We don't believe that it actually matters so long as we have distributed the load, to provide load balancing and fault tolerance.

It may seem intuitive that the hierarchy be constructed around some metric such as Round Trip Time (RTT). So, say, all EEPs in the UK register with an EEP in the UK. This would certainly reduce the network load against, say, a model that had all the registrations made by EEPs in the UK going to Australia. A defence against pathological hierarchies is that the registration time can be measured in minutes or hours not seconds. We can afford such long registration times because we expect the routing *proxylets* to exchange messages at a much higher frequency and form hierarchies based on RTT and bandwidth etc... So a failed node will be discovered rapidly at the routing level. Although it seems like a chicken and egg situation the discovery *proxylets* could request topology information from the routing *proxylets*, to aid in the selection of where a new EEP should register. We also don't want to repeat the message exchanges that will go on in the higher level routing exchanges.

We have considered two other mechanisms for forming hierarchies. The first is a random method. In this scheme a node will only allow a small fixed number of nodes to register with it. Once this number is exceeded any registration attempts will be passed to one of the list of nodes which is currently registered with the node. The selection can be made randomly. If, for example, the limit is configured to be five, the sixth registration request will be provided with the already registered nodes as the new parent EEP. This solution will obviously form odd hierarchies in the sense that they do not map onto the topology of the network. It could however be argued that this method may give an added level of fault tolerance.

The second method that we have been considering is a hierarchy based on domain names, so that the hierarchy maps directly onto the DNS hierarchy. Thus all sites with the domain ".edu.au" all register with the same node. In this case we can use proximity information derived from DNS to build the hierarchy. This scheme will however fail with the domain ".com", where nothing can be implied about location. Also, the node that is accepting registrations for the ".com" domain will be overwhelmed.

We believe that, since registration exchanges occur infrequently, we can choose a hierarchy forming mechanism which is independent of the underlying topology. The more frequent routing exchanges discussed in the next section will map onto the topology of the network and detect any node failures.

We have discussed a hierarchy with a single root. If this proved to be a problem, we could have an infrastructure with multiple roots. But unlike the rest of the hierarchy the root nodes would have to be aware of each other through static configuration, since they would have to pool information in order to behave like a single root.

Messages exchanged by discovery *proxylet*

The types of messages used are *registration messages* and *node messages*. The registration messages are used solely to build the hierarchy. The node messages are used to interrogate the discovery infrastructure.

Version numbers are used to both detect a protocol mismatch, and to trigger the reloading of a new version of the discovery *proxylet*. The *host count* will make it simple to discover the total number of nodes by interrogating the root node.

- Registration request message.

  - Version number.

  - This node's name.

  - Count of hosts registered below this node.

- Registration acknowledgement message,

  sent in response to a registration request message.

  - Version number.

  - Next registration time.

    A delay time before the next registration message should be sent. This timer can be adjusted dynamically as a function of load, or reliability of a node. If a node has many children the timer may have to be increased to allow this node to service a large number of requests. If a child node has never failed to register in the required time, then it may be safe to increase the timeout value.

  - List of nodes to register with,

    used for fault tolerance. Typically the last entry in the list will be the root node.

- Node request message.

  - Version number.

- Node acknowledgement message, sent in response to a node request message.

  - Version number.
  - Host this node registers with.
  - List of backup nodes to register with.

    It is useful to have the list of backup nodes in case the pointer up to the local node fails, while a tree walk is taking place.
  - List of nodes that register with this node.

6.5 Routing exchanges

Once the underlying registration infrastructure is in place this can be used to start routing *proxylets* on the nodes. The process is simple and elegant. A routing *proxylet* can be loaded on any EEP. One routing *proxylet* needs to be started on just one EEP anywhere in the hierarchy. By interrogating the discovery *proxylet* all the children can be found as well as the pointers up the tree. The routing *proxylet* then just starts an instance of itself on every child and on its parent. This process repeats and a routing *proxylet* will be running on every EEP. The routing *proxylet* will also register with the discovery *proxylet* to be informed of changes (new nodes, nodes disappearing, change in parent node). So once a routing *proxylet* has launched itself across the network, it can track changes in the network.

Many different routing *proxylets* can be written to solve various problems. It may not even be necessary run a routing *proxylet* on each node. It may be possible to build up a model of the connectivity of the EEP mesh by only occasionally having short lived, probing *proxylets* running on each cluster. The boundary between having a centralised routing infrastructure against a distributed routing infrastructure can be shifted as appropriate.

We believe that many different routing *proxylets* will be running using different metrics for forming topologies. Obvious examples would be paths optimised for low latency. Or paths optimised for high bandwidth. This is discussed further below.

6.5.1 Connectivity mesh

In a previous section we described a number of *proxylets* that we have already built and are considering building, along with their routing requirements. Some routing decision can be solved satisfactorily by using simple heuristics such as domain name. There will however be a set of services which require more accurate feedback from the routing system.

We believe that some of the more complex routing *proxylets* will have to make routing exchanges along the lines of map distribution MD [19]. A MD algorithm floods information about local connectivity to the whole network. With this information topology maps can be constructed. Routing computations can be made hop by hop. An example may be an algorithm to compute the highest bandwidth pipe between two points in the network. A more centralised approach may be required for application layer multicast where optimal fan out points are required.

In fixed routing architectures each node, by some mechanism, propagates some information about itself and physically connected neighbours. Metrics such as bandwidth or RTT for these links may be included in these exchanges. In the ALR world we are not constrained to using only physical links to denote neighbours. There may be conditions where nodes on opposite sides of the world may be considered neighbours. Also links do not necessarily need to be bidirectional. We expect to use various metrics for selecting routing neighbours. We won’t necessarily be distributing multiple metrics through one routing mesh. We may create a separate routing mesh for each metric.

This is explored further below.

Another important issue which doesn’t usually arise from traditional routing is that if care is not taken, certain nodes may disappear from the routing cloud if a node cannot find a neighbour.

6.5.2 Snooping for network state

The maintenance of network state can be performed via periodic exchanges between routing *proxylets*. While this has merit, it has the disadvantage that the exchanges themselves put load on the network, and therefore impact network performance. While not dismissing this approach, we propose here an alternative approach that uses more implicit routing exchanges.

Proxylets started on EEPs make use of the standard networking API to transmit information. It would be relatively simple to add a little shim layer, such that whenever a networking call is made we can estimate, for example, the bandwidth of a path. Thus bandwidth information derived from service *proxylets*, such as a multicast *proxylet*, can be fed back into the routing infrastructure. An initial proposal for how this might be utilised is now discussed.

6.5.3 Self Organising Application-level Routing - SOAR

In this section we describe a possible approach to building a dynamic routing infrastructure. The requirement is, construct topologies based on various routing metrics [56]. One of the problems that must be overcome is that the underlying topology is not known. Given a large number of nodes a structure must be imposed on these nodes. In order to reduce the scope of the problem neighbouring nodes (by some metric) are formed into clusters. These clusters can themselves then be formed into a heirarchy. It should be borne in mind that nodes may appear and disappear at any time. It is therefore important that connectivity is not lost due to the loss of a node. The maintainance of the SOAR regions is a continuous process. Especially when considering that bandwidth metrics may vary due to network load.

We propose a recursive approach to this, based on extending ideas from RLC[86] and SOT[48], called Self Organised Application-level Routing (SOAR). The idea is that a SOAR does three tasks:

1. Exchanges graphs/maps [19][30] with other SOARs in a region, using traffic measurement to infer costs for links in graphs to re-define regions.

   A region (and there are multiple sets of regions, one per metric), is informally defined as a set of SOARs with comparable link costs between them, and ”significantly” different edge costs out of the region. An election procedure is run within a region to determine which SOAR reports the region graph to neighbour regions. Clearly, the bootstrap region is a single SOAR.

   The above definition of ”significant” needs exploring. An initial idea is to use the same approach as RLC[86] - RLC uses a set of data rates distributed exponentially - typically, say, `10kbps, 56kbps, 256Kbps, 1.5Mbps, 45Mbps` and so on.

   This roughly corresponds to the link rates seen at the edge of the net, and thus to a set of users possible shares of the net at the next ”level up”. Fine tuning may be possible later.

2. SOAR uses measurement of user traffic (as in SOT[48]) to determine the available capacity - either RTCP reports of explicit rate, or inferring the available rate

W_m ; Maximum advertised receive window
RTT ; The Round Trip Time
b ; the number of packets acknowledged by 1 ACK
p ; the mean packet loss probability
B ; the throughput achieved by a TCP flow

Table 6.1: Terms in the Padhye TCP Equation

by modelling a link and other traffic with the Padhye[67] equation provide ways to extract this easily. Similarly, RTTs can be estimated from measurement or reports (or an NTP *proxylet* could be constructed fairly easily). This idea is an extension of the notion proposed by Villamazar[87] using the Mathis[55] simplification, in ”OSPF Optimized Multipath”, $p < (MSS/(BW * RTT))^2$

A more complex version of this was derived by Padhye et al.:

$$B = min(\frac{W_m}{RTT}, \frac{1}{RTT\sqrt{\frac{2bp}{3}} + T_0 min(1, 3\sqrt{\frac{3bp}{8}})p(1+32p^2)}) \tag{6.1}$$

RTT Is estimated in the usual way, if there is two way traffic. $R\bar{T}T_i = RTT_i * alpha + (1 - alpha) * R\bar{T}T_{i-1}$ It can also be derived using NTP exchanges.

Then we simply measure loss probability (p) with a EWMA. Smoothing parameters (alpha, beta for RTT, and loss averaging period for p) need to be researched accurately - note that a lot of applications use this equation directly now[76] rather than AIMD sending a la TCP. This means that the available capacity (after you subtract fixed rate applications like VOIP) is well modelled by this.

Stated simply, given the RTT and loss probability between two nodes it is possible to estimate the bandwidth that a single TCP flow would take. This gives us a bandwidth bound that TCP fair flows should stay within. Another way of considering this is that it also tells us the bandwidth that the next flow has available. The only information that is required is RTT and loss probability. Both of which can be actively or passively measured between nodes.

3. Once a SOAR has established a metric to its bootstrap configured neighbour, it can declare whether that neighbour is in its region, or in a different region - as this continues, clusters will form. The neighbour will report its set of ”neighbour” SOARs (as in a distance vector algorithm) together with their metrics (strictly, we don’t need the metrics if we are assuming all the SOARs in a region are similar, but there are lots of administrative reasons why we may - in any case, a capacity-based region will not necessarily be congruent with a delay-based region. Also, it may be useful to use the neighbour exchanges as part of the RTT measurement).

   The exchanges are of region graphs or maps - each SOAR on its own forms a region and its report is basically like a link state report. We should explore whether the SOAR reports should be flooded within a region, or accumulated as with a distance vector.

   A graph is a flattened list of node addresses/labels, with a list of links for each node, each with one or more metrics.

   Node and Link Labels are in a URL-like syntax, for example `soar://node-id.region-id.soar-id.net` and a Link label is just the far end node label.

   Metrics are `<type, value>` tuples (ASCII syntax). Examples of metrics include:

   - A metric for delay is typically milliseconds

- A metric for throughput is Kbps
- A metric for topological distance is hop count
- A metric for topological neighbour is IP address/mask

As stated above, a SOAR will from time to time discover that a neighbour SOAR is in a different region. At this point, it marks itself as the ”edge” of a region for that metric. This is an opportunity for scaling - the SOARs in a region use an election procedure to determine which of them will act on behalf of the region. The elected SOAR (chosen by lowest IP address, or perhaps at the steiner centre, or maybe by configuration), then pre-fixes the labels with a region id (perhaps made up from date/time and elected SOAR node label.

```
soar://node-id.region-id.soar-id.net/metricname
soar://node-id.region-id.region-id.soar-id.net/metricname
soar://node-id.region-id.region-id.region-id.soar-id.net/metricname
etc
```

6.6 Related Work

Active Networks has been an important area of research since the seminal paper by Tennenhouse et al.[83] (Some people would suggest that this work was preceded by the Softnet [28] work). This paper is based on work in a research project which is more oriented towards active services[31, 7], which has emerged as an important subtopic though the Openarch conference and related events.

In the current work, we are attempting to address problems associated with self-organisation and routing, both internal to Active Services infrastructure, as well as in support of specific user services. To this end we are taking a similar approach to the work in scout[62], specifically the joust system[39], rather than the more adventurous, if less deterministic approach evidenced in the Ants work[90, 15]. Extension of these ideas into infrastructure services has been carried out in the MINC Project[14] and is part of the RTP framework (e.g. Perkins work on RTP quality[68].

Much of the work to date in topology discovery and routing in active service systems has been preliminary. We have tried to draw on some of the more recent results from the traditional (non active) approaches to these two sub-problems, and to this end, we have taken a close look at work by Francis[29], as well as the nimrod project[19].

Our approach is trying to yield self-organising behaviour as in earlier work[48][15][93], as we believe that this is attractive to the network operator as well as to the user.

There have been a number of recent advances in the area of estimation of current network performance metrics to support end system adaption, as well as (possibly multi-path) route selection, e.g. for throughput, there is the work by Mathis[55], Padhye[67] and Handley[76], and for multicast[86], and its application in routing by Villamizar[87]. more recently, several topology discovery projects have refined their work in estimating delays, and this was reported in Infocom this year, for example, in Theilman[84], Stemm[82], Ozdemir[66] and Duffield[24].

Chapter 7

Conclusion

Current networking implementation choices for applications are limited in two dimensions. Firstly: options available to the builder of a network application are limited. For a two party communication, if reliability is required, the fastest way to build an application is to build it using a reliable transport such as TCP - even if the application's requirements map poorly to the services provided by TCP. The second dimension in the problem is that of protocol distribution. Two endpoints communicating with each other must use the same version of a protocol (or one must support backward compatibility). It takes time for communication stacks to be deployed. It also takes time for enhancements or bug fixes to be incorporated and deployed. In any significant protocol there must therefore be backward compatibility. Typically protocols employ an option negotiation stage. During this phase both ends converge on a mutually acceptable interaction set. There are rarely epochs where it is possible to say protocol A version 1 will disappear and henceforth only protocol A version 2 will be in use.

In the *slogin* (2) and MMD (3) chapters it is demonstrated that designing network protocols to map onto an application's requirements improves efficiency. In the *slogin* case it provides better interactivity on long delay lossy links. The problem remains that significantly more work is required to build an application level protocol (ALP) than to use traditional methods. We are led to the observation that if it was made easier to write application specific protocols then they would be used. Initial work was done in terms of decomposing protocols; ALPs [32]. The eventual goal was to provide building blocks from which protocols could be built. Other members of the Hipparch project [5], worked on solving this problem using automatic protocol composition using Esterel [23]. This approach of providing a library of building blocks was not pursued. A library of building blocks would have been a viable approach to have taken.

A more efficient place to attack the problem seemed to be that of deployment. However, what would be deployed? Thus the problem of simplifying the building of ALPs still exists. The traditional way is that a protocol is defined; an example of a protocol could be the Simple Mail Transfer Protocol (SMTP) [75]. As the protocol became popular each new mail product required a new implementation of the protocol. The SMTP protocol is designed to run over any reliable transport protocol. An ALP could be designed that had the same functionality as SMTP. This ALP could run over UDP. Carefully designed small mail messages could be delivered using only a two packet exchange. Clearly more packets would be required if any packets were lost. In contrast SMTP running over TCP there will be at least three packets exchanged to setup the TCP connection. There will also be at least three packets to tear down the connection. The SMTP part of the protocol will add another few packets to the exchange. It is therefore more efficient in terms of network utilisation to use an ALP, however, there is still the problem of increased implementation complexity.

The decision was taken to tackle the problem from another angle. If it was possible to produce one implementation of a protocol that was shared by all applications then the extra implementation effort may be justified. This is not a new idea. Traditionally such a packaging of code is called a library. However, a library only partially solves the problem. It may force the choice of what language the rest of the application is written in. It will almost certainly not work across operating system platforms. At the time this problem was being considered a new programming language appeared on the scene. This programming language was Java [9]. Java seemed to offer a solution to the library problem. A Java program compiled down to platform independent byte codes and Java virtual machines (JVMs), Java interpretors were available across a range of platforms. It seemed that it should be possible to develop ALPs in Java that could be portably incorporated into applications. The vision was, that in the future, an implementation of a protocol would be available for download from a standard bodies server. Applications would be written such that when a protocol stack was required; one could be download on demand. Once downloaded, as it was comprised of portable platform independent byte codes, it could be integrated into the application. This integration could be either directly into an application or as a separate process that the application interacts with.

Multiple benefits could be derived from such a scheme. Firstly: only one implementation of a protocol would be required. The extra effort required in implementing such a protocol would therefore be justified. Secondly: a single protocol implementation downloaded on demand would solve all possible version control problems. When a bug is fixed in the protocol stack or an addition made it could be made available in place of the older stack. It would not be this simple if functionality was to be added to a protocol stack as the application may require knowledge of these new options.

In the JavaRadio chapter (4) initial work pursuing these ideas is described. Firstly any doubts about the performance of Java had to be dispelled. Java is an interpreted language, it also uses a memory allocator based on garbage collection. Both of these factors could render it unsuitable for building protocol stacks. In order to test both ALP ideas and performance concerns, a real time audio application was chosen. It was shown that Java could meet the performance constraints of a real time audio playout. The groundwork was also laid for ALP stacks.

They were not however implemented. Dynamically linking an ALP stack into an application turned out to be problematic. The issue was in the API between the application and the ALP. Should the API be generic enough to capture the semantics of all possible network stacks? Perhaps there should be APIs per application domain. Initially it was believed that it should be possible to capture all possible interactions by just modelling the API on standard networking APIs. An API that supported the standard (connection setup, data transfer phase and connection tear down) and then enhanced with mechanisms to dynamically attach the stack. A number of problems were encountered when considering RTP, the stack which was developed for JavaRadio. As developed The RTP stack performed transforms on the data by applying a codec. Input data presented in PCM form to the stack could be transformed into LPC before transmission. A generic API might then support a mechanism for transforms to be performed on data. A more generic API would support a chain of data transforms (e.g. compression followed by encryption by . . .). However, in RTP a more complex problem was encountered. It is possible to send RTP audio with redundancy [69]. The basic idea is to send previous audio samples with the present audio sample. With a large enough playout buffer at the receiver, if the previous audio sample was lost the next sample can be used. Multiple levels of redundancy can be performed. Each level of redundancy goes back in the packet history. Typically the redundant samples would use codecs with higher compression than the primary sample. A packet could therefore carry not only the current piece of data but previous pieces of data. However, the encoding of the data would be different for each transmission. This is one example of why a generic API would be difficult to develop. Another problem is paradoxically with versioning. One of the intended goals was to do away with possible version mismatches. If a new feature is added to the ALP the application would still need to be modified in some cases to utilise this feature. One possible solution to this problem may have been to place a configuration GUI with the ALP. New options could then be presented directly to the user. The conclusion is not that it is not possible to define a generic API, rather that it would be difficult and prone to the pitfalls described above. A generic API may emerge from the reliable multicast building block [91] or similar work.

Another solution to loading ALPs into end applications is to load them into the network. This mechanism was termed “Application Level Active Networking” (ALAN), chapter 5. The ALAN ideas are implemented in the form of a package *funnelWeb*. The ideas from chapter 4 were taken and integrated into this new framework. The streaming audio ideas were fitted into a WWW context. For the streaming audio example a mechanism was shown for deployment which required no changes to the end systems except for minor configuration changes were required. It was shown that new application specific protocols could be deployed into the network. The two original goals, one to show that benefit can be derived from application level protocols and the second to ease the deployment of such protocols, are met in *funnelWeb*. *Proxylets* can be written and made dynamically available. Only one implementation of a *proxylet* is required for any particular task. *Proxylets* are loaded by reference. If for whatever reason a *proxylet* is changed, the old version is replaced by the new version. Any subsequent attempts to use the proxylet will automatically get the new *proxylet*. There are flaws in such a scheme, a bug can be deployed as quickly as a fix. Incremental deployment is not possible; this is not necessarily bad.

The *funnelWeb* infrastructure demonstrates a useful concept. However, one more component was required. For all the initial experiments the location of EEPs (Execution Environment for *Proxylets*) had to be known beforehand. What was required was a mechanism for using the *funnelWeb* infrastructure dynamically. If an ALAN style system were to be deployed then Application Layer Routing (ALR) would be necessary. In chapter 6 the ALR scheme is described. With this final component it is possible to use the *funnelWeb* infrastructure as a complete and usable system.

7.1 Future Work

A number of projects in the research community are currently using *funnelWeb*: ALPINE [1], RADIOACTIVE [4] and ANDROID [2]. There has also been some interest in the Internet Engineering Task Force (IETF) community to use the *proxylet* concept in Open Pluggable Edge Services (OPES)[3]. This serves to validate the ideas presented.

For a *funnelWeb* style infrastructure to be truly interesting a wide scale global would be necessary. This would allow for many new services to be built and deployed. No incentive currently exists for a site to run an EEP. If a user could be charged (Chapter 5.8) for running a *proxylet* then wider deployment would occur. This might also be the incentive for ISPs to put EEPs in core of the network.

In Chapter 6, some preliminary work in application layer routing is described.

Scalable routing protocols need to be built within this framework such as application layer multicast.

Appendix A

Glossary

ACK Acknowledgement

ADPCM Adaptive Pulse Code Modulation

AIMD Additive Increase Multicative Decrease

ALF Application Layer Framing

ALP Application Layer Protocol

ALR Application Layer Routing

API Application Programming Interface

BT British Telecom

CBQ Class Based Queuing

DLPI Data Link Provider Interface

DNS Domain Name System

DPS Dynamic Proxy Servers

DVMRP Distance Vector Multicast Routing Protocol

EEP Execution Environment for Proxylets

EWMA Exponentially Weighted Moving Average

FPGA Field Programmable Gate Array

FreeBSD Free Version of the Berkeley Software Distribution

Funnelweb Application Level Active Networking Infrastructure

GUI Graphical User Interface

HTML Hypertext Markup Language

HTTP Hypertext Transfer Protocol

IETF Internet Engineering Task Force

ILP Integrated Layer Processing

IP Internet Protocol

ISDN Integrated Services Digital Network

ISI Information Sciences Institute

ISP Internet Service Provider

INRIA The French National Institute For Research In Computer Science And Control

JAR Java Archive

JDK Java Development Kit

JIT Just in Time Compiler

JVM Java Virtual Machine

Jeeves Sun Microsystems web system

LAN Local Area Network

LPC Linear Predictive Coding

LZS Lempel-Ziv standard compression

MBONE Multicast Back Bone

MMD Multicast Mail Delivery

MTU Maximum Transmission Unit

NACK Negative Acknowledgement

NAT Network Address Translator

OPES Open Pluggable Edge Services

OSPF Open Shortest Path First

PCB Process Control Block

PCM Pulse Code Modulation

PDA Personal Digital Assistant

PSTN Public Switched Telephone Network

RFC Request For Comment

RLC Receiver driven Layered Congestion control

RMI Remote Method Interface

RPC Remove Procedure Call

RTCP RTP Control Protocol

RTP Real Time Protocol

RTT Round Trip Time

SOAR Self Organising Application-level Routing

SPAM Unsolicited Email

SRM Scaleable Reliable Multicast

SSH Secure Shell

SYN Synchronize

Slogin Simple Login

TCL Tool Command Language

TCP Transmission Control Protocol

TTL Time to Live

UCL University College London

UDP User Datagram Protocol

URG TCP Urgent Pointer

URL Universal Resource Locator

UTS University of Technology Sydney

VM Virtual Machine

VOIP Voice over IP

WAP Wireless Application Protocol

WML Wireless Markup Language

XML Extensible Markup Language

YAAT Yet Another Audio Tool

Appendix B

Flakeway

The *Flakeway* tool was built in order to test *slogin* and MMD applications.

All networks will lose packets. The probability of a packet being lost can vary, but packets will always be lost. Therefore network protocols and Application Level Protocols (ALPs) need to be robust against packet losses.

When ALPs are being tested and debugged it is useful to be able to simulate loss. It is possible to simulate loss in a number of ways. The application itself could be modified to drop packets. Tests could be performed on real networks in the hope that with enough testing all problems will be discovered. A program such as Dummynet [77] which is installed in a FreeBSD kernel could be used to drop packets. Another possibility is to use *flakeway*, which predates Dummynet.

Design goals:

- Deterministic and repeatable behaviour
- Driven from real packet data traces
- No Kernel modifications
- Simple to insert in packet flow

Value	Action
> 0	delay in milliseconds
0	forward packet no delay
< 0	drop packet

Figure B.1: Possible actions

For the rest of this Appendix we will describe the operations and implementation of *flakeway*. At the time that *slogin* (Chapter 2) was being written a method was required to test *slogin*. As discussed above code could have been put into *slogin* to simulate loss. This was not done, it would have added extra code to *slogin* unnecessarily. The code would not have available to other applications. It was decided to build a *flakeway* outside the application.

The main feature required from *flakeway* was deterministic behaviour. This would allow tests to be repeated and make it possible to craft test scenarios. It would also allow real traffic traces to be used directly as input to *flakeway*.

Another requirement was that no application changes should be required when using *flakeway*. A mechanism was devised to place *flakeway* in the path of a packet flow. This will be described later in the implementation section.

In order to provide deterministic behaviour *flakeway* is table driven. Tables are associated to destination addresses. A packet that enters *flakeway* and does not have an associated table is ignored. Figure B.1 shows the three actions that a packet can be subjected to. A packet can be: dropped, forwarded immediately or delayed.

There are two types of action tables that an incoming packet can match. A destination address is associated with one of these two types of action table.

Both types of action tables contain lists of actions as described in Figure B.1. The action tables are traversed sequentially and when the end of the table is reached the

Action
0
-1
1000

Figure B.2: Example Packet count table

Time period	Action
1000	0
1000	-1
1000	1000

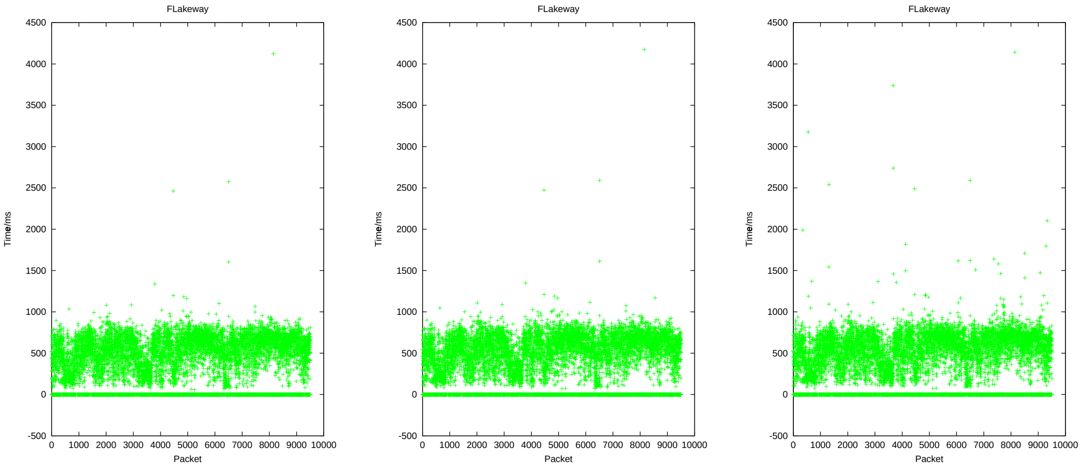
Figure B.3: Example Time sequence table

process starts again. The tables behave like circular lists.

The first type of table is a “Packet count table” Figure B.2 is an example of such a table. Each packet that arrives is matched against the current entry. The packet is then subjected to the action in the current entry. The next entry then becomes the current entry in readiness for the next packet. When the next entry is off the end of the list then the current entry becomes the first entry in the table. To work through the example in Figure B.2; the first packet is forwarded with no delay, the second packet is dropped, the third packet is delayed for 1000ms, the fourth packet restarts the process.

The second type of table is a “Time sequence table” Figure B.3 is an example of such a table. In the packet count examples advances through the table were made due to packet arrival. In the time sequence advances are made as a function of time period. Each time the time period expires the next action in the table is used. In Figure B.3; any packet arriving in the first 1000ms will be forwarded, in the next 1000ms packets will be dropped, in the next 1000ms they will be delayed by 1000ms. This scheme allows the emulation of periodic outages or delays.

A very simple experiment was perfomed, a ping was run from UCL to INRIA in



(a) Original

(b) Packet Number

(c) Time Sequence

Figure B.4: Ping Trace. UCL - INRIA

Sophia Antipolis. The ping output was then used to generate packet number and time sequence traces. Pings were then run through *flakeway*. The traces were then plotted in Figure B.4. It can be observed that original trace and the packet number trace look almost identical. This is to be expected, whatever befell the original ping packet in terms of loss or delay is emulated by *flakeway* does. In the case of the time sequence trace the two graphs are not so correlated. This can perhaps be explained by the time bases between the pings and the table slipping.

Flakeway is a useful tool for being able to perform repeated experiments and perhaps emulate the behaviour of a real network, from packet traces. One limitation that will be discussed in the next section is that it could not handle large loads, as it is implemented in user space.

B.1 Implementation

A major design constraint was that it should be possible to use *flakeway* without altering the applications under test. The question is how to interpose oneself into the traffic

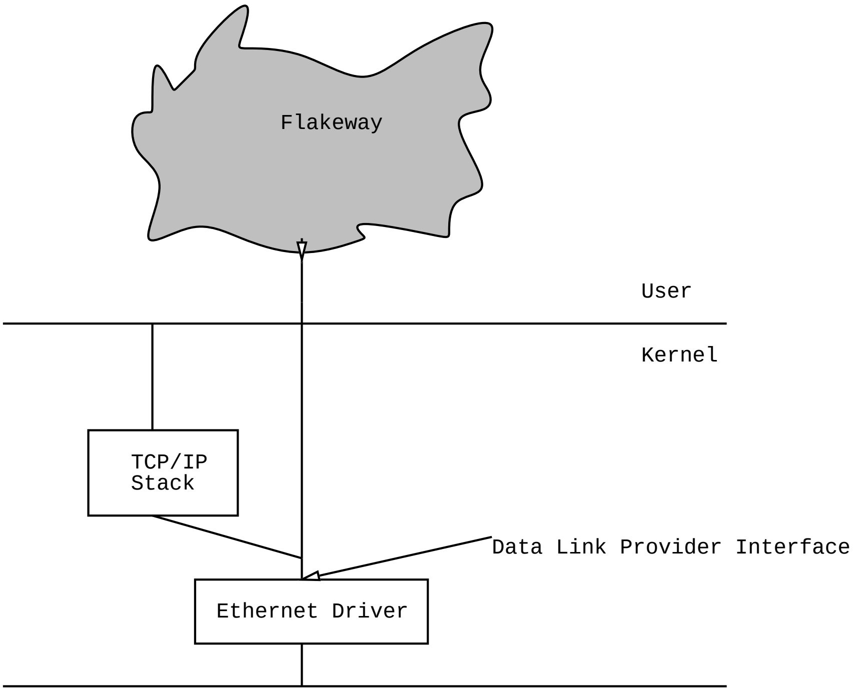


Figure B.5: Flakeway workstation

flow. The solution taken by Dummynet was to place a module in the FreeBSD kernel. In previous CBQ [88] work on Solaris, streams modules were used to get control of packets. The CBQ approach would also have required two network interfaces, which *flakeway* didn't A major design goal had been no kernel modifications.

The work was being done on Solaris 2.4 and this operating system provided a Data Link Provider Interface (*DLPI*) [85]. The *DLPI* allows the transmission and reception of MAC frames bypassing the usual protocol handling. Standard *DLPI* implementations have a minor flaw, they do not pass up or allow the setting of the MAC header. Fortunately Sun have extended *DLPI* to allow the transmission and reception of the MAC header.

Flakeway is a user space program which uses the *DLPI* interface to gather and send packets (see Figure B.5).

Flakeway will receive a copy of all packets that arrive on this interface. *Flakeway*

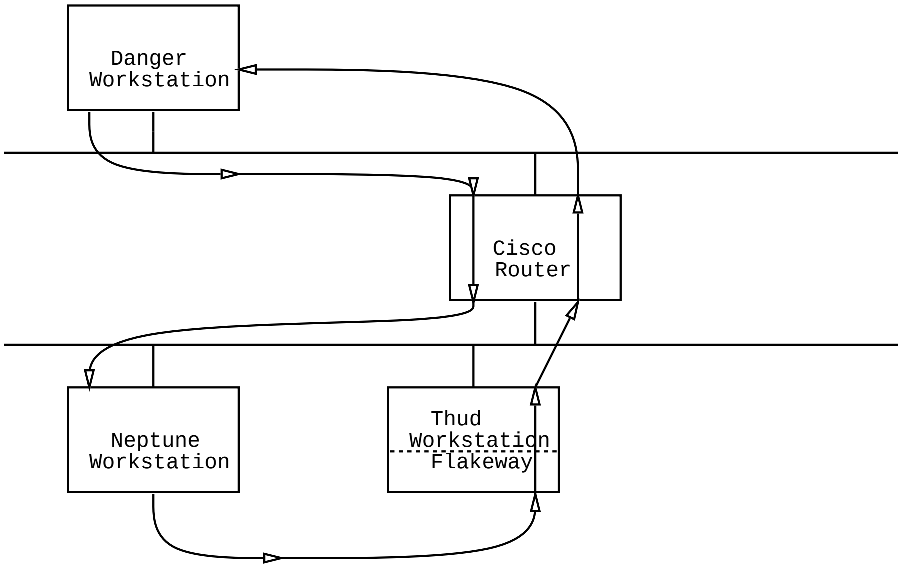


Figure B.6: Network configuration

can also construct and send a whole MAC frame.

Consider Figure B.6, the trick is to get the host “neptune” to send packets to “thud”. Packets received by *flakeway* on thud destined for danger will be processed. Those packets selected to be forwarded will be sent to “danger”.

We face two problems; firstly how do we get packets from “neptune” to *flakeway*, secondly how does *flakeway* send these packets to “danger”?

The first part of the problem is to set a host route from “neptune” to be “danger” to be via “thud”. All packets destined for “danger” will now traverse “thud” the flakeway machine. If IP forwarding is disabled then the host stack won’t attempt any forwarding. Packets for “danger” will arrive at “thud” and *flakeway* will process them. If a received packet is to be forwarded by *flakeway* it rewrites the MAC and transmits it.

In order help find potential mis-configurations rather than set the gateway address to “thud”, it was set to a special *flakeway* address. After testing if a bad host route was not cleared, forwarding entries with *flakeway* addresses would hopefully stand out. This means that *flakeway* has to respond to ARP requests for its special address.

B.2 Conclusions

Flakeway is run in user space so the flow which it is manipulating must be sending at a relatively low data rate. For testing *slogin* and audio samples for JavaRadio the data rate was low enough for *flakeway* to be used.

A further enhancement rather than to delay or drop might be to corrupt the packet. Being able to duplicate packets may also be a useful feature.

Dummynet is able to rate limit a flow to simulate a bottleneck. It is also able to drop packets probabilistically. *Flakeway* is able to do neither. Adding probabilistic dropping would be simple, rate limiting would high data rates would not be possible from user space.

Flakeway turned out to be extremely useful for testing *slogin* and JavaRadio, especially as explicit drop patterns could be programmed.

Appendix C

TCP Evolution

Some examples of why TCP is not always the appropriate choice for network applications.

Since TCP [72] was specified in 1981 networks have evolved. This evolution has meant that for periods of time TCP has not been an optimal medium for some applications. As the networks have evolved applications using TCP and TCP itself has had to evolve. Mismatches between networks and TCP have become apparent and then addressed. Also there have been problems with application and TCP mismatches [65].

In way of example the following two sections give examples of: application to TCP mismatches and TCP to network mismatches.

It could be postulated, that in some of these cases, an application layer protocol (ALP), tightly matched to an applications requirements, would have fared better than applications implemented using TCP.

C.1 Application modifications

The application Netscape is commonly used for web browsing. The protocol that Netscape uses is HTTP.

C.1.1 Netscape - multiple streams

The protocol used to retrieve content from a web server is HTTP. In initial versions of the HTTP protocol 1.0 a separate TCP connection was required for every URL. If for example a web page contained many images then a TCP connection would be required for each image. A connection per image. When faced with this problem the Netscape implementors chose to open multiple simultaneous connections. This increased the user's satisfaction as web pages would be downloaded faster than the previous single connection scheme. Multiple connections however are extremely unfriendly to the network. TCP's slow start algorithm is in some sense defeated as the congestion window is multiplied by the number of connections attempted. The multiple TCP flows are competing with each other and causing more load on the network than a single flow.

C.1.2 HTTP - single connection

A problem with the HTTP 1.0 protocol was that a single TCP connection is used for each request. So in order to retrieve a page a connection is created a request is made and the end of the page is marked by the connection being torn down. This is inefficient in a number of ways. If many pages are retrieved from one server then multiple connection setup and connection tear down packets have to be exchanged. Each separate TCP connection will need to go through a congestion control phase to reach a stable operating rate (slow start). A longer running connection is better able to stabilise and better utilise the available bandwidth.

With later versions of the HTTP [25] protocol all transactions with a web server can be performed over one persistent connection.

C.2 Implementation modifications

As well as applications having to be modified due to an inappropriate interaction between TCP and the application, modifications have been suggested and implemented in TCP. This is not an exhaustive list but shows modifications which have been required with time.

A large set of modifications and a requirement for backward compatibility may cause interaction and implementation problems in the future.

C.2.1 Transaction TCP T/TCP

T/TCP [13] is an attempt to support a Remote Procedure Call (RPC) still interaction for TCP. T/TCP does not require multiple exchanges of setup packets and is designed to cache flow state information.

HTTP over T/TCP would be more efficient than over TCP.

C.2.2 Large Windows

As networks become faster it became apparent that the 64K maximum window size in TCP is not sufficient to fully utilise a Gigabit pipe. Extensions have been proposed to allow larger window sizes [42].

C.2.3 Selective Acknowledgement - SACK

TCP's standard acknowledgement scheme, is a cumulative acknowledgement. The loss of a single segment in a sequence of segments cannot be described to the peer. When an acknowledgement is received the only information that can be garnered is that segments upto this acknowledgement number have been received. If ten segments are sent and the fifth segment is lost, the sender on seeing the acknowledgement has no way of knowing only one segment has been lost and could very well retransmit all the unacknowledged segments.

An optional selective acknowledgement [54] was therefore added to TCP. A later addition was made to the scheme to allow for the signalling of duplicate received segments [26].

C.2.4 Congestion Control

In order to stop the congestion collapse of the Internet various congestion control mechanisms are typically in most TCP implementations [41].

C.2.5 Unreliable Delivery

TCP provides reliable delivery. Some real time streaming applications such as audio or video can continue to function even if some packets in the stream are lost. A suggestion has been made to modify a TCP receiver to send ACK's for packets which have not been received. If an application selects this option for a real time stream a large delay will not build up as packets are retransmitted.

Path MTU discovery

Different parts of the Internet have different Maximum Transmission Unit sizes. The Internet Protocol suite mandates a minimum transmission size of 576 octets. This size is a lot smaller than say an Ethernet maximum frame size which is 1500 octets. Traditional TCP implementations would use the maximum transmission size if two hosts were on the same subnet and revert to the lower size of 576 octets if the hosts were on different subnets. Two hosts on different subnets with a transmission size of greater than 576 would not fully utilise the link.

So a modification was made to TCP which allowed TCP [61] to discover the maximum available link size on a path and use it.

This is an extremely valuable modification. Unfortunately the mechanism to determine the optimum packet size on a link is not available to UDP based ALF applications.

Appendix D

RC4 Implementation

Figure D.1 and D.2 are the header file and C source file of the RC4 implementations taken from slogin. They are presented here in full to demonstrate the simplicity of the algorithm. In figure D.2 the function *rc4next,* is called for each byte in the data stream.

```
#define BSIZE 256

struct rc4 {
        u_char sbox[BSIZE];
        u_char key[BSIZE];
        int i;
        int j;
};

struct rc4 *rc4init(u_char *key, int len);
u_char rc4next(struct rc4 *rc4);
```

Figure D.1: rc4.h

```
#include "rc4.h"

static
void
swap(u_char *p1, u_char *p2)
{
        u_char temp;

        temp = *p1;
        *p1 = *p2;
        *p2 = temp;
}
struct rc4 *
rc4init(u_char *key, int len)
{
        struct rc4 *rc4;
        int i, j;

        rc4 = malloc(sizeof(struct rc4));
        if(0 == rc4)
                return 0;

        for(i = 0; i < BSIZE; i++)
                rc4->sbox[i] = i;

        for(i = 0; i < BSIZE; i++)
                rc4->key[i] = key[i % len];

        for(j = i = 0; i < BSIZE; i++) {
                j = (j + rc4->sbox[j] + rc4->key[i]) % BSIZE;
                swap(&rc4->sbox[i], &rc4->sbox[j]);
        }

        rc4->i = rc4->j = 0;

        return rc4;
}

u_char
rc4next(struct rc4 *rc4)
{
        int t;

        rc4->i = (rc4->i + 1) % BSIZE;
        rc4->j = (rc4->j + rc4->sbox[rc4->i]) % BSIZE;
        swap(&rc4->sbox[rc4->i], &rc4->sbox[rc4->j]);
        t = (rc4->sbox[rc4->i] + rc4->sbox[rc4->j]) % BSIZE;

        return rc4->sbox[t];
}
```

Figure D.2: rc4.c

Bibliography

[1] Alpine - application level programmable inter-network environment. <URL:http://www.cs.ucl.ac.uk/research/alpine/>. BT Sponsored.

[2] Android - active network distributed open infrastructure development. <URL:http://www.cs.ucl.ac.uk/research/android/>. Framework V project.

[3] Opes - open pluggable edge services. <URL:http://www.ietf-opes.org/>. IETF Working Group.

[4] Radioactive - realising adaptive distributed internet operations on active networks. <URL:http://www.cs.ucl.ac.uk/research/radioactive/>. DARPA Sponsored.

[5] High performance protocal architecture (hipparch). <URL:http://www.docs.uu.se/hipparch/>, 1994 - 1996. Esprit Basic Project.

[6] Akamai. <URL:http://www.akamai.com/>.

[7] Elan Amir, Steven McCanne, and Randy Katz. An active service framework and its application to real-time multimedia transcoding. *Computer Communication Review*, 28(4):178–189, September 1998.

[8] S. Armstrong, A. Freier, and K. Marzullo. Multicast transport protocol. Request for Comments 1301, Internet Engineering Task Force, February 1992.

[9] Ken Arnold and James Gosling. *The Java Programming Language*. Addison Wesley, 1996.

[10] Tim Berners-Lee. The world-wide web initiative. In *Proceedings of the International Networking Conference (INET)*, pages DBC–2 – DBC–4, San Francisco, California, August 1993. Internet Society.

[11] Bluetooth.
<URL:http://www.bluetooth.com/>.

[12] R. Braden. Requirements for internet hosts – communication layers. Request for Comments 1122, Internet Engineering Task Force, October 1989.

[13] R. Braden. T/tcp – tcp extensions for transactions functional specification. Request for Comments 1644, Internet Engineering Task Force, jul 1994.

[14] R. Caceres, N.G. Duffield, J. Horowitz, and D. Towsley. Multicast-based inference of network-internal loss characteristics. *IEEE Transactions on Information Theory*, 45(7), 1999.

[15] Maria Calderon, Marifeli Sedano, Arturo Azcorra, and Cristian Alonso. Active network support for multicast applications. *ieeenet*, 12(3):46–52, May 1998.

[16] Pei Cao, Jin Zhang, and Kevin Beach. Active cache: Caching dynamic contents on the web. In *In Proceedings of IFIP International Conference on Distributed Systems Platforms and Open Distributed Processing (Middleware ’98)*, pages 373–388, 1998.

[17] Stephen Casner and Stephen Deering. First IETF Internet audiocast. *Computer Communication Review*, 22(3):92–97, July 1992.

[18] Jo-Mei Chang and N. F. Maxemchuk. Reliable broadcast protocols. *ACM Transactions on Computer Systems*, 2(3):251–273, August 1984.

[19] J. Noel Chiappa. Nimrod.
<URL:http://ana-3.lcs.mit.edu/ jnc/nimrod/docs.html>.

[20] David D. Clark and David L. Tennenhouse. Architectural considerations for a new generation of protocols. In *ACM Special Interest Group on Data Communication*, pages 200–208, Philadelphia, Pennsylvania, September 1990. IEEE. Computer Communications Review, Vol. 20(4), Sept. 1990.

[21] R. V. Cox and P. Kroon. Low bit-rate speech coders for multimedia communication. *IEEE Communications Magazine*, 34(12):34–40, December 1996.

[22] W. Diffie and M.E. Hellman. Multiuser cryptographic techniques. In *Proceedings fo AFIPS National Computer Conference*, pages 109–112, 1976.

[23] C. Diot, R. de Simone, and C. Huitema. Protocol development using esterel. *Journal for High Speed Networks, Special issue on HIPPARCH*, jun 1996.

[24] Nick Duffield and Francesco Lo Presti. Multicast inference of packet delay variance at interior network links. In *Infocom*. IEEE, 2000.

[25] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, and T. Berners-Lee. Hypertext transfer protocol – HTTP/1.1. Request for Comments 2068, Internet Engineering Task Force, January 1997.

[26] S. Floyd, J. Mahdavi, M. Mathis, and M. Podolsky. An extension to the selective acknowledgement (sack) option for tcp. Request for Comments 2883, Internet Engineering Task Force, jul 2000.

[27] Sally Floyd, Van Jacobson, Ching-Gung Liu, Steven McCanne, and Lixia Zhang. A reliable multicast framework for light-weight sessions and application level framing. *Computer Communication Review*, 25(4):342–356, October 1995.

[28] R. Forchheimer and J. Zander. Softnet - packet radio in sweden. In *Proc. of AMRAD Conference*, Washington, DC, 1981.

[29] Paul Francis. Yallcast.
<URL:http://www.yallcast.com/>.

[30] Paul Francis. *Addressing in Internetwork Protocols*. PhD thesis, University College London, September 1994.

[31] Michael Fry and Atanu Ghosh. Application level active networking. *Computer Networks*, 31(7):655–667, April 1999.

[32] A Ghosh and J Crowcroft. Some lessons learned from various alf and ilp applications. *Australian Computer Journal*, 28(2), May 1996.

[33] A. Ghosh, M. Fry, and J. Crowcroft. An architecture for application layer routing. In *Active Networks, Second International Working Conference, IWAN 2000, Tokyo, Japan, October 16-18, 2000, Proceedings*, volume 1942 of *Lecture Notes in Computer Science*, pages 71–86. Springer, 2000.

[34] Atanu Ghosh and Michael Fry. Funnelweb.
<URL:http://dmir.it.uts.edu.au/projects/alan/funnelWeb-2.0.1.html>.

[35] Atanu Ghosh and Michael Fry. Javaradio: an application level active network. In *Third International Workshop on High Performance Protocol Architectures (HIP-PARCH '97*, June 12-13 1997.

[36] Atanu Ghosh, Michael Fry, and Glen MacLarty. An infrastructure for application level active networking. *Computer Networks*, 36(1):655–667, June 2001.

[37] V. Hardman, M.A. Sasse, M. J. Handley, and A. Watson. Reliable audio for use over the internet. In *Proceeding of INET 95*, (1995.

[38] Vicky Hardman, Martina Angela Sasse, and Isidor Kouvelas. Successful multiparty audio communication over the internet. *Communications of the ACM*, 41(5):74–80, May 1998. Association for Computing Machinery.

[39] John J. Hartman, Peter A. Bigot, Patrick Bridges, Brady Montz, Rob Piltz, Oliver Spatscheck, Todd A. Proebsting, Larry L. Peterson, and Andy Bavier. Joust: A platform for liquid software. *IEEE Computer*, 32(4):50–56, April 1999.

[40] Inktomi. Broadcast overlay architecture. <URL:http://www.inktomi.com/products/media/pdfs/whtpapr.pdf>.

[41] V. Jacobson. Congestion avoidance and control. *ACM Computer Communication Review; Proceedings of the Sigcomm '88 Symposium in Stanford, CA, August, 1988*, 18, 4:314–329, 1988.

[42] V. Jacobson, R. Braden, and D. Borman. Tcp extensions for high performance. Request for Comments 1323, Internet Engineering Task Force, may 1992.

[43] Van Jacobson and Steve McCanne. vat - LBNL audio conferencing tool. <URL:http://www-nrg.ee.lbl.gov/vat/>, July 1992.

[44] M. Kadansky, D. Chiu, J. Wesley, and J.Provino. Tree-based reliable multicast (tram). Internet draft, Internet Engineering Task Force. Work in Progress, 1998.

[45] B. Kantor. Bsd rlogin. Request for Comments 318, Internet Engineering Task Force, April 1991.

[46] Brian W. Kernighan and Dennis M. Ritchie. *C Programming Language*. Prentice Hall, 2nd edition, 1989. ISBN 0-13-110362-8.

[47] G. Kessler and S. Shepard. A primer on internet and tcp/ip tools. Request for Comments 1739, Internet Engineering Task Force, December 1994.

[48] Isidor Kouvelas, Vicky Hardman, and Jon Crowcroft. Network adaptive continuous-media applications through self organised transcoding. In *nossdav*, pages 241–255, Cambridge, England, July 1998.

[49] Danny Lange and Mitsuru Oshima. *Programming and Deploying Java[tm] Mobile Agents with Aglets [tm]*. Addison Wesley, 1998.

[50] Danny Lange and Mitsuru Oshima. Seven good reasons for mobile agents. *Communications of the ACM*, 42(3):99–89, March 1999.

[51] Kurt Lidl, Josh Osborne, and Joseph Malcolm. Drinking from the firehose: multicast USENET news. In *usenixw*, pages 33–45, San Francisco, California, January 1994.

[52] G. MacLarty and M. Fry. Policy-based content delivery: An active network approach. In *Web Caching and Content-delivery Workshop, Lisbon Portugal*, 2000.

[53] Glen MacLarty and Michael Fry. Policy-based content delivery: an active network approach. In *The 5th International Web Caching and Content Delivery Workshop*, May 22-24 2000. Lisbon, Portugal.

[54] M. Mathis, J. Mahdavi, S. Floyd, and A. Romanow. Tcp selective acknowledgment options. Request for Comments 2018, Internet Engineering Task Force, oct 1996.

[55] Matt Mathis, Jeffrey Semke, Jamshid Mahdavi, and Teunis Ott. The macroscopic behavior of the TCP congestion avoidance algorithm. *Computer Communications Review*, 27(3):67–82, July 1997.

[56] Laurent Mathy, Roberto Canonico, and David Hutchison. An overlay tree building control protocol. In Jon Crowcroft and Marcus Hofmann, editors, *Proceedings of the Third International COST264 Workshop (NGC 2001)*, number LNCS 2233, pages 76–87, London, UK, November 2001. Springer-Verlag.

[57] P. M. Melliar-Smith, L. E. Moser, and V. Agrawala. Membership algorithms for asynchronous distributed systems. In *International Conference on Distributed Computing Systems (ICDCS*, May 1991.

[58] D. Meyer. Administratively scoped ip multicast. Request for Comments 2365, Internet Engineering Task Force, jul 1998.

[59] G. Minshall, Y. Saito, J. Mogul, and B. Verghese. Application performance pitfalls and tcp's nagle algorithm, 1999.

[60] P. Mockapetris. Domain names - concepts and facilities. Request for comments, Internet Engineering Task Force, nov 1987.

[61] J. Mogul and S. Deering. Path mtu discovery. Request for Comments 1063, Internet Engineering Task Force, nov 1990.

[62] A. Montz, D. Mosberger, S. O'Mealley, L. Peterson, T. Proebsting, and J. Hartman. Scout: A communications-oriented operationg system. Technical Report 94-20, Department of Computer Science, The University of Arizona, 1994.

[63] John Nagle. Congestion control in ip/tcp internetworks. Request for Comments 896, Internet Engineering Task Force, January 1984.

[64] Real Networks.
<URL:http://www.real.com/>.

[65] S. O'Malley and L. Peterson. Tcp extensions considered harmful. Request for comments, Internet Engineering Task Force, oct 1991.

[66] Volkan Ozdemir, S. Muthukrishnan, and Injong Rhee. Scalable, low-overhead network delay estimation. In *Infocom*. IEEE, 2000.

[67] Jitendra Padhye, Victor Firoiu, Don Towsley, and Jim Kurose. Modeling TCP throughput: A simple model and its empirical validation. *Computer Communications Review*, 28(4):303–314, September 1998.

[68] C Perkins. Rqm.
<URL:http://www-mice.cs.ucl.ac.uk/multimedia/software/rqm/>.

[69] C. Perkins, I. Kouvelas, O. Hodson, V. Hardman, M. Handley, J.C. Bolot, A. Vega-Garcia, and S. Fosse-Parisis. Rtp payload for redundant audio data. Request for Comments 2198, Internet Engineering Task Force, sep 1997.

[70] J. Postel. User datagram protocol. Request for comments, Internet Engineering Task Force, August 1980.

[71] J. Postel. Internet control message protocol. Request for Comments 972, Internet Engineering Task Force, October 1981.

[72] J. Postel. Transmission control protocol. Request for comments, Internet Engineering Task Force, September 1981.

[73] J. Postel. Tcp and ip bake off. Request for Comments 1025, Internet Engineering Task Force, sep 1987.

[74] Jon Postel. Telnet protocol. Request for Comments 318, Internet Engineering Task Force, April 1972.

[75] Jonathan B. Postel. Simple mail transfer protocol. Request for Comments 821, Internet Engineering Task Force, aug 1982.

[76] Reza Rejaie, Mark Handley, and Deborah Estrin. An end-to-end rate-based congestion control mechanism for realtime streams in the internet. In *Infocom*, New York, March 1999.

[77] L. Rizzo. Dummynet: A simple approach to the evaluation of network protocols. *Computer Communications Review*, 27(1), January 1997.

[78] Bruce Schneier. *Applied Cryptography Second Edition: protocols, algorithms, and source code in C.* Wiley, 1996. ISBN: 0-471-11709-9.

[79] H. Schulzrinne. RTP profile for audio and video conferences with minimal control. Request for Comments 1890, Internet Engineering Task Force, January 1996.

[80] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson. RTP: a transport protocol for real-time applications. Request for Comments 1889, Internet Engineering Task Force, January 1996.

[81] Alan Schwartz. *Managing Mailing Lists*. O'Reilly & Associates, March 1998. ISBN: 156592259X.

[82] Mark Stemm, Srinivasan Seshan, and Randy H. Katz. A network measurement architecture for adaptive applications. In *Infocom*. IEEE, 2000.

[83] D. L. Tennenhouse and D. J. Wetherall. Towards an active network architecture. *Computer Communication Review*, 26(2):5–18, April 1996.

[84] Wolfgang Theilmann and Kurt Rothermel. Dynamic distance maps of the internet. In *Infocom*. IEEE, 2000.

[85] OSI Work Group Unix International. Data link provider interface specification.

[86] L. Vicisano, L. Rizzo, and J. Crowcroft. TCP-Like congestion control for layered multicast data transfer. In *Infocom*, San Francisco, Californi, March/April 1998.

[87] Curtis Villamizar. Ospf optimized multipath (ospf-omp). Internet draft. work in progress, Internet Engineering Task Force, March 1998. draft-ietf-ospf-omp-00.

[88] I Wakeman, A Ghosh, J Crowcroft, V Jacobson, and S Floyd. Implementing real time packet forwarding policies using streams. In *Usenix*, pages 71–82, New Orleans, Louisiana, January 1995.

[89] A. Westine and J. Postel. Problems with the maintenance of large mailing lists. Request for Comments 1211, Internet Engineering Task Force, mar 1991.

[90] David Wetherall, Ulana Legedza, and John Guttag. Introducing new internet services: Why and how. *IEEE Network*, 12(3):12–19, May 1998.

[91] B. Whetten, L. Vicisano, R. Kermode, M. Handley, S. Floyd, and M. Luby. Reliable multicast transport building blocks for one-to-many bulk-data transfer. Request for Comments 3048, Internet Engineering Task Force, January 2001.

[92] Brian Whetten, Simon Kaplan, and Todd Montgomery. A high performance totally ordered multicast protocol. research memorandum, August 1994.

[93] Nicholas Yeadon. *Quality of Service Filters for Multimedia Communications*. PhD thesis, Computing Department, Lancaster University, Bailrigg, Lancaster, LA1 4YR, U.K, May 1996.

[94] Tatu Ylonen. Ssh — secure login connections over the internet. In *Proceedings of the Sixth Usenix Unix Security Symposium*, pages 37–42, jun 1996.