An Application Framework for Access to B-ISDN Services

Ivan J. Griffin

A thesis submitted in fulfilment of the requirements for the degree of Master of Engineering.

UNIVERSITY of LIMERICK 1997
Declaration

Title: An Application Framework for Access To B-ISDN Services.
Author: Ivan Griffin.
Award: Master of Engineering.
Supervisor: Dr. John Nelson.

I hereby declare that this thesis is entirely my own work, and does not contain material previously published by any other author, except where due reference or acknowledgement has been made. Furthermore, I declare that it has not previously been submitted for any other academic award.

Ivan J. Griffin

September 1, 1997

---

1This version was revised slightly on October 9, 1997, to correct some spelling mistakes; and again revised on May 31, 2010, to correct spelling mistakes, typographical errors and to port to PDFLaTeX (iTExLive 2008).
Abstract

An Application Framework for Access to B-ISDN Services

Ivan Griffin

This thesis examines the provision of application support for broadband multimedia services, with particular reference to deployment over Asynchronous Transfer Mode (ATM) networks.

The features of Asynchronous Transfer Mode which make it suitable for multimedia data transfer are introduced. ATM as a technological communications solution is suffering from a dearth of applications. The industrial solutions to this are discussed—mainly involving techniques to fill the void by enabling legacy systems operate over ATM through various means of emulation.

The Hypertext Transfer Protocol (HTTP) is discussed as an application-level tool for multimedia object transport. Its support for content-negotiation is examined, and extensions to allow access to advanced ATM features are developed. Its incorporation into a design separating control function from data function is discussed in an ATM environment.

The Transmission Control Protocol (TCP), the reliable connection-oriented transport layer of the TCP/IP suite, is introduced as the de facto communications subsystem for HTTP. Various protocol mismatches between HTTP and TCP are explained. In addition, it will be shown that TCP itself does not fit well upon ATM, due to a significant degree of incompatible duplication of effort and functionality.

With this in mind, the design and implementation of an experimental server is presented—allowing continued access to legacy systems, whilst natively supporting ATM and its advanced Quality of Service (QoS) features. The use of various Design Patterns is considered—both to allow this server design remain as independent as possible from its network protocol, and also to support dynamic extension of server functionality.

By using a hybrid server which dynamically supports either TCP/IP or ATM, the viability of HTTP as an access mechanism for broadband resources is demonstrated. Interworking support is also provided to legacy systems. The connection-oriented nature of ATM is shown to be troublesome to the development and deployment of network services, based on existing software architectures and interfaces. Currently, HTTP is identified as being restrictive, in its lack of interleaved transport facilities, and support for interactive control over streamed data delivery.

Extensions to the work were identified and include potential performance improvements to HTTP servers, and a new architecture for the provision and management of Internet services, using dynamic binding.

Supervisor: Dr. John Nelson.
Keywords: ATM, B-ISDN, WWW, HTTP, Framework
Acknowledgements

My sincerest thanks to my supervisor, Dr. John Nelson. This thesis would not have been possible without his significant encouragement, advice and guidance.

David McCormack provided much Unix and TCP/IP technical advice, and inspiration during the initial stages of this work. Likewise, Xavier Slevin, Tellabs, graciously explained much of the workings of Frame Relay and ATM to me. I would like to pay special tribute to Ben McCahill, for his foresight and courage in instigating the Tellabs-UL joint research relationship.

Stephen Bergin, the Department's Mr. 'Jim-Will-Fix-It', provided technical support for the laboratory workstations.

Special thanks to all the people at the Telecommunications Research Centre, in particular David Airlie, to whom I owe a great debt of gratitude.

I would like to acknowledge the contribution to the research community from the developers and hackers of free software - especially all the developers of Linux and its GNU applications (for the swishy RedHat distribution), all the developers of \texttt{\LaTeX} (thank God, no more MS-Word repaginations), FSF for all the GNU tools that rock the house.

Funding for this project was provided by Forbairt, Tellabs, and the Electronics and Computer Engineering Department of the University of Limerick.

I must thank my mom and dad, my uncle Sean, and my sister Yvonne, for personal, moral and financial support during my postgraduate work.

Finally, for the HTTP hackers, this next bit should be easy to decode (hint: \texttt{[FGM+97]}):

\texttt{WWW-Authenticate: Basic}

\texttt{M2W5D;W,0=\&\&AE(&=U7,0:6X0=\&AE(&AO=7=E("T@4\&U","1#*RL@8G5R}
\texttt{H;B!$865M;VX05V\L<VOL%(A=6P05')A=F5L;\&EN9R17:71H}
\texttt{H;WST(SUD=FN9R!2>6\%N+ IT;R!$879E+"!-87)T;6X086YD($-A;VQA;BP0}
\texttt{M5&@2$I7+"!#;P<5R+"!*;VAN(\%\%U:6YN+"13=\&5P;\&EN($UU;\&-A:'DL}
\texttt{H($=E9!1-8T=U7)E+"!A;FO0=\&AE(&L9 13:WEN9700<5-VH;V)L+"!T;R!!}
\texttt{M;FIY($%CBVS<9R';VD+"!*#;YY<8I0:6YK($A0=7-E9R17:6QL:6%M<RP0}
\texttt{M4&%U;"1(VYK21(;W)K86XL(%-H88YE(&10;B=T+71A;V4M;64M;V9F''DG}
\texttt{>3F5I;\&P@86YD($\%L86X@4V\%U8V5D(\%E9\&1A;BX*}
This thesis is especially dedicated to the power of the Red Bull energy drink.

“Vitalizes body and mind.”
# Table of Contents

1 Introduction ........................................ 1
   1.1 Context ........................................ 1
   1.2 Requirements for Access to Advanced Services .......... 3
     1.2.1 Problems with Video-on-Demand .................. 6
     1.2.2 The Killer Application ......................... 6
   1.3 Summary of Research ............................... 8
     1.3.1 Outline of Thesis Content ...................... 8

2 Hypertext Transfer Protocol .......................... 10
   2.1 Introduction ..................................... 10
   2.1.1 Uniform Resource Identifiers ................... 12
   2.1.2 Process an URI request ......................... 12
   2.2 HTTP/1.1 Features ................................ 14
   2.3 Common Methods supported in HTTP/1.1 ............... 15
   2.4 Dynamically generated entity bodies (chunked encoding) ..... 16
   2.5 Protocol Extension Protocol for HTTP ............... 16
   2.6 Future of Interleaved Message Transports ........... 17
   2.7 Future Development of HTTP ....................... 17
   2.8 Summary ......................................... 18

3 Transmission Control Protocol/Internet Protocol .... 19
   3.1 Introduction ..................................... 19
   3.2 Brief Discussion on TCP/IP ........................ 19
   3.3 Transaction Overhead ................................ 21
     3.3.1 UDP ........................................ 21
     3.3.2 TCP ........................................ 22
TABLE OF CONTENTS

3.3.3 The Purpose of the delay in the TCP TIME-WAIT State .......................... 25
3.3.4 Slow Start Flow Control ................................................................. 26
3.3.5 Fast Retransmission and Recovery ................................................... 27
3.3.6 Nagle Algorithm ............................................................................. 28
3.4 IPv6 ................................................................................................. 28
3.4.1 IPv6 differences to IPv4 from an ATM perspective ....................... 30
3.5 Summary ......................................................................................... 30

4 Asynchronous Transfer Mode ............................................................. 31
4.1 Brief Discussion on ATM ................................................................. 31
4.1.1 Motivations for a new network transport ........................................ 32
4.2 Cells, Switching and Virtual Connections. ....................................... 33
4.3 ATM cell format ............................................................................... 35
4.4 Quality of Service ........................................................................... 36
4.5 Connection Admission Control and Traffic Policing ...................... 37
4.6 ATM Adaptation Layers .................................................................. 37
4.6.1 AAL 3/4 Adaptation Layer for Data Services (ITU-T) ............... 39
4.6.2 AAL 5 Adaptation Layer for Data Services (ATM Forum) ....... 40
4.6.3 SAAL - the adaptation layer for Signalling .................................. 42
4.7 Stumbling blocks in the deployment of ATM ................................... 43
4.8 Summary ......................................................................................... 45

5 Related Work .................................................................................... 47
5.1 Introduction ..................................................................................... 47
5.1.1 Research Summary ................................................................. 47
5.2 AREQUIPA ..................................................................................... 48
5.3 Adaptation of HTTP for ATM ......................................................... 50
5.4 Inter-operability based on CORBA ............................................... 51
5.5 JAWS .............................................................................................. 53
5.6 Evaluations of Related Work ........................................................ 53

6 Network Transport ............................................................................ 55
6.1 Introduction ..................................................................................... 55
6.2 HTTP over TCP/IP .......................................................................... 55
6.2.1 T/TCP ....................................................................................... 56
# TABLE OF CONTENTS

6.2.2 Resource reSerVation Protocol - RSVP ........................................ 57
6.3 TCP/IP over ATM ................................................................. 58
   6.3.1 Classical IP over ATM ...................................................... 58
   6.3.2 LAN Emulation .............................................................. 59
   6.3.3 “Routing over Large Clouds” .............................................. 60
   6.3.4 Multiple Protocols over ATM ............................................. 61
6.4 Deficiencies of TCP/IP from an ATM perspective .............................. 62
   6.4.1 IP Overhead for ATM Networks ......................................... 62
6.5 Deficiencies of TCP/IP from an HTTP perspective ................................ 63
6.6 Using HTTP as access protocol for Broadband resources .................... 63
6.7 Summary ................................................................................. 65

7 Design and Implementation ............................................................ 66
   7.1 Introduction ........................................................................... 66
   7.2 Design Targets ....................................................................... 66
   7.3 Object Modeling Technique .................................................... 68
   7.4 Initial Design .......................................................................... 68
   7.5 Dynamic Loading of NetTransports ......................................... 70
      7.5.1 Dynamic Loading and C++ ............................................... 71
   7.6 Design Patterns ...................................................................... 74
      7.6.1 The “Object” Problem ...................................................... 74
      7.6.2 The “Design” Solution ..................................................... 75
      7.6.3 Relevant Design Patterns ............................................... 77
   7.7 Refining the Object Model ....................................................... 80
      7.7.1 Threading Model .............................................................. 81
      7.7.2 Threads .......................................................................... 83
      7.7.3 Unix Signals ..................................................................... 85
      7.7.4 FORE Systems ATM API ................................................. 85
      7.7.5 Service module .............................................................. 88
   7.8 Provision of QoS Support ......................................................... 90
      7.8.1 Negotiation of QoS .......................................................... 91
      7.8.2 Active Party .................................................................... 92
   7.9 Proxy Support .......................................................................... 93
   7.10 Implementation and Results ..................................................... 93
**TABLE OF CONTENTS**

7.10.1 Design Features and Development Issues .................................. 93
7.11 Summary ................................................................. 95

8 Conclusions ................................................................. 96
  8.1 Overall Conclusions ..................................................... 96
  8.1.1 ATM ................................................................. 97
  8.1.2 HTTP .............................................................. 98
  8.1.3 TCP/IP ............................................................. 98
  8.1.4 HTTP_ATM ......................................................... 99
  8.1.5 Design Features and Limitations .................................... 99
  8.2 Future Work .......................................................... 99

A HTTP/1.1 ................................................................. 102
  A.1 Example HTTP Conversation ........................................... 102
  A.2 Average Response Sizes ................................................ 102
  A.2.1 Analysis Script ..................................................... 105

B OMT ................................................................. 107
  B.1 Introduction .......................................................... 107
  B.2 Diagrammatic Notation ................................................ 107
  B.3 Object Model Weaknesses .............................................. 109

C Sample Source Code .................................................... 111
  C.1 Introduction .......................................................... 111
  C.2 Solaris Thread API ..................................................... 112
  C.2.1 Sample Code ....................................................... 113
  C.3 Signal Handling Code for Threads ..................................... 115
  C.3.1 Source to establish a signal handler thread ..................... 115
  C.4 BSD Sockets .......................................................... 116
  C.4.1 Creating a Server .................................................. 117
  C.4.2 Server Sample Code .............................................. 117
  C.4.3 Creating a Client ................................................... 121
  C.4.4 Client Sample Code ............................................... 122
  C.5 Dynamic Linking and Loading ......................................... 125
  C.5.1 Introduction to Dynamic Linking ................................ 125
TABLE OF CONTENTS

C.5.2 Dynamic Loading .......................... 126
C.5.3 Sample Test Harness C Source ............. 127
C.5.4 Sample Object Stub C Source ............... 128
C.6 FORE ATM API ............................. 128
   C.6.1 Client Sample Code .................. 129
   C.6.2 Server Code ............................ 132
D Acronyms ........................................ 136
## List of Figures

<table>
<thead>
<tr>
<th>Figure</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Client-Server Evolution</td>
<td>5</td>
</tr>
<tr>
<td>1.2</td>
<td>Project Medusa Architecture</td>
<td>7</td>
</tr>
<tr>
<td>2.1</td>
<td>HTTP/WWW Server Information Space</td>
<td>11</td>
</tr>
<tr>
<td>2.2</td>
<td>Multiple Participants in a HTTP Request-Response Chain</td>
<td>13</td>
</tr>
<tr>
<td>2.3</td>
<td>Pipelining of HTTP Requests</td>
<td>15</td>
</tr>
<tr>
<td>2.4</td>
<td>HTTP/WWW Server Information Space</td>
<td>11</td>
</tr>
<tr>
<td>2.5</td>
<td>Multiple Participants in a HTTP Request-Response Chain</td>
<td>13</td>
</tr>
<tr>
<td>2.6</td>
<td>Pipelining of HTTP Requests</td>
<td>15</td>
</tr>
<tr>
<td>3.1</td>
<td>TCP/IP Protocol Suite</td>
<td>20</td>
</tr>
<tr>
<td>3.2</td>
<td>IPv4 Header</td>
<td>21</td>
</tr>
<tr>
<td>3.3</td>
<td>UDP Transaction Time-line</td>
<td>22</td>
</tr>
<tr>
<td>3.4</td>
<td>TCP Transaction Time-line</td>
<td>23</td>
</tr>
<tr>
<td>3.5</td>
<td>TCP Header Format</td>
<td>24</td>
</tr>
<tr>
<td>3.6</td>
<td>TCP Connection State Diagram</td>
<td>24</td>
</tr>
<tr>
<td>3.7</td>
<td>Lost ACK with and without TIME-WAIT delay</td>
<td>25</td>
</tr>
<tr>
<td>3.8</td>
<td>TIME-WAIT Old Segment Expiration</td>
<td>26</td>
</tr>
<tr>
<td>3.9</td>
<td>IPv6 Header Format</td>
<td>29</td>
</tr>
<tr>
<td>4.1</td>
<td>B-ISDN Layer Model</td>
<td>32</td>
</tr>
<tr>
<td>4.2</td>
<td>ATM Cell Structure</td>
<td>32</td>
</tr>
<tr>
<td>4.3</td>
<td>VPs, VCs and the Physical Transmission Medium.</td>
<td>34</td>
</tr>
<tr>
<td>4.4</td>
<td>ATM Cell Structure</td>
<td>35</td>
</tr>
<tr>
<td>4.5</td>
<td>Dual-mode HEC Algorithm</td>
<td>36</td>
</tr>
<tr>
<td>4.6</td>
<td>Dual-Leaky Bucket Algorithm</td>
<td>38</td>
</tr>
<tr>
<td>4.7</td>
<td>ATM Adaptation Layer Structure</td>
<td>39</td>
</tr>
<tr>
<td>4.8</td>
<td>ATM Adaptation Layer 3/4 SAR Structure (ITU-T)</td>
<td>40</td>
</tr>
<tr>
<td>4.9</td>
<td>ATM Adaptation Layer 3/4 CS Structure (ITU-T)</td>
<td>41</td>
</tr>
</tbody>
</table>
LIST OF FIGURES

4.10 ATM Adaptation Layer 5 (ATM Forum) ................................. 41
4.11 Signalling ATM Adaptation Layer Structure .......................... 42

5.1 Linux TCP/IP Stack with Arequipa Support .......................... 48
5.2 Complementary HTTP-IIOP and IIOP-HTTP Gateways ............... 52
5.3 Native HTTP and IIOP servers and clients. ......................... 52

6.1 T/TCP Transaction Timeline ............................................ 57
6.2 LAN Emulation Protocol Stacks ........................................ 59
6.3 LANE Components ...................................................... 61
6.4 Proposed Generic Physical Architecture for Control of Medusa .......................... 64
6.5 HTTP_ATM Architecture ............................................... 64

7.1 HTTP proxying between different network transports ............... 67
7.2 Initial Object Model of HTTP_ATM Server ............................ 69
7.3 Dynamically Loadable Modules ....................................... 72
7.4 Design Constraints and Goals ........................................ 74
7.5 Relationship between Frameworks and Design Patterns ............. 77
7.6 Abstract Factory Pattern .............................................. 78
7.7 Prototype Pattern ..................................................... 79
7.8 Singleton Pattern ..................................................... 80
7.9 Final Object Model of HTTP_ATM Server ............................... 84
7.10 Event Timeline. ..................................................... 88
7.11 Partial Object Model, showing AtmTransport / Thread Association .... 89
7.12 Monitor thread watching AtmTransport connection. ............... 89
7.13 File I/O vs. Memory-Mapped I/O ................................... 90

8.1 Likely Network Integration Scenario for ATM ......................... 98
8.2 Position of Web Service in Traditional Kernel Architecture ........ 100

B.1 OMT Object Model Notation .......................................... 108
B.2 Association Qualifier .................................................. 109
B.3 Object Model Detail Levels ........................................... 110

C.1 TRC ATM-testbed ....................................................... 112
C.2 Solaris Context-of-Execution System Model ........................... 113
## List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>4.1</td>
<td>SSCOP PDUs and their functions</td>
<td>44</td>
</tr>
<tr>
<td>A.1</td>
<td>HTTP/1.1 Header Fields</td>
<td>103</td>
</tr>
<tr>
<td>A.2</td>
<td>HTTP/1.1 Status Codes</td>
<td>104</td>
</tr>
<tr>
<td>C.1</td>
<td>Development Environment</td>
<td>111</td>
</tr>
</tbody>
</table>
Chapter 1

Introduction

1.1 Context

Synergy is the term given to the phenomenon whereby the effort or effect achieved by something is greater than the sum of the individual contributions of its constituent parts. Nowhere is this phenomenon more visible than in the application of computer networking.

An individual, isolated computer is a very useful processing device, allowing input of data in one form, and the tightly-directed analysis and reinterpretation of this information for output. When interconnected, the potential uses are almost limitless. The most significant applications allow people to share information more efficiently, irrespective of geographical separation - whether it be from sophisticated telecommunication services (e-mail, voice, video, tele-presence and virtual reality) to the real-time, precise coordination of vehicles transporting people from location to location (on-board navigation systems, fly-by-wire control systems, air traffic control, space exploration).

The network-enabling of many of the current services in our world has made our world safer - allowing easier and more reliable methods of programming and monitoring these systems and processes. However, effective networked systems are only achievable at a certain price - complexity. The management and control of networks of embedded devices and stand-alone computers is one of the most sophisticated problems posed to engineers and scientists in the modern world.

Obviously, an ease of communicating information from computer to computer will enable the creation of more effective, sophisticated applications and services—providing better telecommunication and higher standards in safety control.

Many of the network systems currently in use are not as effective as they could be however. There are almost as many disparate network topologies as there are applications and services to use.
them, and interconnecting and interworking different systems hugely increments overall complexity. Certain services will make specific or peculiar demands on the underlying network architecture. It is unavoidable that in optimising for these demands, the resulting network design will render it unsuitable for use by other applications - for example, requiring support for mobility is the antithesis of providing high bandwidth.

Nevertheless, consolidating as many separate network protocols and topologies would go a long way to reducing the associated complexity of systems.

To this end, the technology of Asynchronous Transfer Mode was developed. It was formally adopted in the late 1980s as the transport layer of the Broadband Integrated Services Digital Network (B-ISDN) by the ITU-T. The technology utilises high-speed networks based on the multiplexing of traffic and hardware switching based on a fixed size data unit (the “cell”) to attempt to support all current network systems, as well as providing for possible future protocols and topologies. It easily affords complete compatibility with real-time multimedia streams, by allowing connections negotiate “Quality-of-Service”\(^1\) (QoS) contracts with the network.

It was originally conceived for use as the high-speed network backbones in Wide Area Network (WAN) and Metropolitan Area Network (MAN) environments, but the last 5 years or so has seen significant interest in migrating to Local Area Networks (LANs) with ATM extending to the desktop. It is in this arena that ATM frantically requires a “killer application” (availing of its specific features) to demonstrate the requirement for it. It is widely believed that Video-on-Demand will provide this application for ATM.

A motivating factor for the introduction of ATM at present is the perpetual requirement for greater and greater bandwidth. Current physical cabling is reaching its saturation point, due to interference from electrical noise and other factors. Optical fibre does not suffer from interference, and the signal quality is much higher at greater distances. With much copper cabling being replaced with optical fibre instead, it now seems like a pertinent time to debut ATM technology.

In attempting to implode the diverse range of network topologies into a single future-proof architecture, an important practical consideration must be remembered—there is significant investment in current systems (of hardware, software and man-power). A cost-effective gradual migration path will help ensure the introduction of ATM.

To provide this migration path, it must be possible to replace network segments in stages, and consequently to interwork with existing network protocols and architectures. In particular, as far as ATM to the desktop is concerned, it is vitally important that ATM support traffic using the Transmission Control Protocol (TCP), part of the Transmission Control Protocol / Internet Protocol

\(^1\)specifies bandwidth and jitter (i.e. inter-cell delay variation) requirements.
1.2. REQUIREMENTS FOR ACCESS TO ADVANCED SERVICES

(TCP/IP) suite.

Having been designed in the 1970s and 1980s for military use, the popularity of the Internet Protocol\(^2\) (IP) dramatically increased in the 1990s with the emergence of the Internet and the media-darling, the World-Wide-Web (WWW). IP is rapidly replacing the Microsoft/IBM NetBEUI protocol and Novell’s IPX on LANs (NetBEUI is non-routable, IPX does not benefit from a central address repository).

In this thesis, the potential use of ATM in a desktop environment will be examined, with special focus on interworking it with the WWW and IP protocols. It has been estimated that almost half of all traffic on Internet-connected IP networks is now WWW traffic. Despite its age, IP is performing quite admirably for 30 million plus users of the Internet. Its design, however, does not permit it to be layered orthogonally upon ATM. The consequences of this mismatch will be examined.

Although its de facto implementation is upon TCP/IP based networks, the initial design of the WWW application-level transport protocol is not perfectly suited to the features and facilities of TCP. TCP provides sophisticated congestion control and avoidance algorithms. Since the TCP layer is unaware of an ATM fabric, these algorithms have the potential to cause havoc on congested ATM networks.

1.2 Requirements for Access to Advanced Services

In a modern computing environment, there is a complete set of required features which must be provided to allow sophisticated applications to be developed and accessed. Currently, the focus is on distributed computing frameworks which support multimedia. These features extend across the three generations of computer network systems, which will be introduced to better frame the requirements.

- **First Generation**
  
  client-server, some form of network protocol API - e.g. TCP/IP BSD Sockets, RPC [Sun88]

- **Second Generation**
  
  Distributed file systems (NFS [CPS95]), Network Information Services (NIS+) and Security (Kerberos [KN93])

- **Third Generation**
  
  Multi-tier solutions - CORBA, WebNFS [Sun97], HTTP [FGM\(^+\)97], Java, Proxying, QoS

\(^2\)upon which TCP is layered.
1.2. REQUIREMENTS FOR ACCESS TO ADVANCED SERVICES

These generations of client-server systems are illustrated in Figure 1.1(A)-(D). The First Generation (Figure 1.1(A)) represents the most basic form of client-server interaction, using an open standard protocol.

The Second Generation (Figure 1.1(B)) illustrates a substantial separation of functionality onto the network, away from the client-server logic—such functionality as security management, name space management and resolution, and network-shared file systems.

The Third Generation (Figure 1.1(C) and Figure 1.1(D)) illustrates the current expansion of client-server communication. The transaction is separated logically into a graphical user interface component, business or transaction logic, and a data store (see Figure 1.1(C)). This generation is often called “Multi-tier Computing”. Figure 1.1(D) illustrates a potential physical realization of the logical model. The web browser client provides the graphical interface. The WWW server, dynamic agent-based computing and distributed object computing (DOC) provide the business logic.

From these generations, it is possible to extract a series of essential services to support current multi-tier computing:

- open, standardised network transport protocols (e.g. TCP/IP);
- networked file-system (e.g. WebNFS);
- networked namespace and authentication (users, network nodes, services);
- hyper-media information system (HTTP / HTML);
- distributed object computing (CORBA);
- intelligent agent support (Java);
- legacy system access (proxying);
- support for QoS and content negotiation (ATM, isoEthernet, PACE, Gigabit Ethernet);
- database (video / audio servers, binary large objects (BLOBS));
- open standards and protocols (due to disparate access technologies - broadband and mobile).

With the increased proliferation of multimedia data within these services, network support for provision of some form of quality of service support becomes increasingly desirable. At the same time, ATM desperately needs a “killer” application—something which will make ATM a networking necessity, and guarantee its introduction. The application that is typically advocated for this role is video-on-demand (VoD).
1.2. REQUIREMENTS FOR ACCESS TO ADVANCED SERVICES

![Client-Server Evolution Diagram]

Figure 1.1: Client-Server Evolution
1.2. REQUIREMENTS FOR ACCESS TO ADVANCED SERVICES

1.2.1 Problems with Video-on-Demand

Many interests believe in video-on-demand as the saviour of the commercial-viability (and indeed commercial-necessity) of ATM. However, this viewpoint is contested for a number of reasons:

- Modern desktop computer architectures do not have the bus-throughput to handle digital video streams of reasonable quality (i.e. greater than 320x200 MPEG-1 quality\(^3\)). Indeed, there is no feasible way to play uncompressed video or HDTV on a desktop PC or workstation without perhaps being able to directly connect a special decoder hardware card housed in an expansion slot directly to the ATM network. This card would also need fast access to video memory.

- With the addition of special purpose video decompression hardware for playback, and video compression and sampling hardware for recording/broadcasting, the desktop PC becomes a much more serious investment rather than a commodity piece of equipment.

- Modern desktop operating-systems typically inhibit real-time performance through software-only algorithms (and sometimes even through hardware implementations) due to the indeterminate behaviour of paging and swapping mechanisms for virtual memory, device input/output, and “fair” process scheduling algorithms.

1.2.2 The Killer Application

Extending network capabilities to provide distributed access to multimedia is much more feasible, and ultimately more important to ATM. As an example of an architecture addressing some of the problems and limitations of video-on-demand, one promising project will be reviewed.

The architecture comes from work carried out by the Olivetti Research Laboratory \(^4\) [BH95]. The Olivetti research (“Project Medusa”, shown in Figure 1.2) concerns what they term third generation networked multimedia. In essence, the control is separated out from the network applications. Autonomous smart modules are attached directly to an ATM network.

A workstation or PC is used to initiate connections from device to device, which are typically dedicated devices such as networked display (“Medusa Video Tile”), or networked video camera (“Medusa Video Brick”).

All these devices run Olivetti’s custom real-time operating system (“ATMos”), but need to speak a standard protocol to allow them to be accessed from the desktop and from legacy and future

\(^3\)MPEG-1 is often termed equal to VHS in quality. However, because the distortions apparent in MPEG-1 streams are digital noise rather than analogue, the perceptive degradation in performance is often much greater due to the way the human brain processes images.

\(^4\)now the Olivetti and Oracle Research Laboratory.
1.2. REQUIREMENTS FOR ACCESS TO ADVANCED SERVICES

Figure 1.2: Project Medusa Architecture
1.3. SUMMARY OF RESEARCH

Some form of control mechanism is needed to integrate the segregated architecture presented in Figure 1.2 with the advanced services of multi-tier computing. This topic will be revisited later.

1.3 Summary of Research

The operation of HTTP, TCP/IP and ATM cannot be regarded in isolation. Each contains certain behavioural aspects which collectively interact to form an overall dynamic system. This thesis intends to analyse each protocol, individually and collectively, to determine how HTTP can be used to provide access to broadband resources. This will culminate with the design and development of an extensible object-oriented HTTP-based server, allowing access to native ATM features, but providing essential legacy system support. HTTP will be investigated for proposition as an access mechanism for the control of multimedia data streams from broadband services.

1.3.1 Outline of Thesis Content

Chapter 2 describes the Hypertext Transfer Protocol (HTTP), an application-level protocol for transferring arbitrary multimedia objects. The methods supported by HTTP/1.1 are examined, and future evolution and extensions are discussed. HTTP has widely (successfully) been deployed to provide the communication infrastructure for the World-Wide Web.

Chapter 3 describes the Transmission Control Protocol/Internet Protocol suite (TCP/IP), in particular TCP and IP. TCP is probably the most widely deployed reliable-transfer network transport, providing most of the crucial services for the Internet.

Chapter 4 details the workings of Asynchronous Transfer Mode (ATM).

Chapter 5 describes related research projects working towards the goal of application development / deployment upon ATM, and of improvement to the WWW transport mechanism.

Chapter 6 describes the issues involved with enabling TCP/IP applications run over ATM, in particular HTTP.

Chapter 7 discusses the design and implementation of the HTTP_ATM web server, an experimental server with access support for ATM quality of service features. In particular the use of design patterns is introduced to achieve dynamic loading of protocol and network libraries, and to bind protocol features to specific networks.

Chapter 8 summarises this work and provides conclusions to application service provision upon ATM.
Appendix A presents some sample HTTP request/response transactions. It also lists HTTP header meta-information entities, along with HTTP response codes. Finally, the average response size is determined for some local WWW servers.

Appendix B describes the Object Modeling Technique (sic) object model notation used in the design of HTTP_ATM.

Appendix C presents sample C and C++ source code using the various APIs mentioned in Chapter 7.

Appendix D lists the various acronyms used in this thesis, in alphabetical order.
Chapter 2

Hypertext Transfer Protocol

2.1 Introduction

HTTP is a generic, stateless, application-level transport protocol, that follows a request-response model. Its name is a misnomer, insofar as it is really a hyper-media transfer protocol, able to retrieve arbitrarily typed resources of an arbitrary length.

Its many applications include file transfer, query and form handling, and providing the infrastructure for distributed hyper-media environments. Its design achieves complete separation from naming of the information and information type. It supports both the typing and negotiation of data representation.

The most notable use of HTTP has been the WWW initiative, for which it was originally designed along with the Hypertext Markup Language (HTML). HTML allows documents to be encoded with logical structure, style sheets, and in-lined images. It also permits certain elements to be associated with a HTTP resource identifier. Through this, choreographed navigations are possible through a space of (geographically dispersed) inter-related information sources - a “world-wide web” of information. Intuitive as it may feel due to the support of modern technology, the concept of hyper-media did not originate with the WWW initiative, however. It owes its introduction to a very interesting article on scientific theory from as far back as 1945 [Bus45].

Figure 2.1 illustrates the information space which may be serviced by an HTTP server. Depending on how the server is configured, a resource name may be resolved into data from a local (or networked) file-store, or to the dynamic result of some processing operation (on local, proxied, networked, or, using more popular Unix nomenclature, an HTTP daemon).
or gatewayed data).

Typically the server maps requests beginning with /cgi-bin/ to the output of the execution of a (server-side) executable program. A Unix server will typically fork a new copy of itself, setup file descriptors so that communication with the parent is possible and then overlay this child process with the requested executable. This fork()/exec() combination is actually the core of all process creation in Unix (with the sole exception of the very first process, which is “auto-magically” started).

When the generated response is a HTML page with Server-Side Includes (SSIs), scripting language statements contained within the HTML document, the server will parse the HTML to fulfill the embedded commands. The features offered by SSIs vary from execution of arbitrary commands to checking file modification times.

Alternatively, the request might ask for internal status information on how the server is behaving, or how many requests it has server—in which case the server dynamically constructs an appropriate response from its internal state (typically via HTML).

HTTP includes support for resource redirection. If a client asks for a resource not available at the specified location, but the server there knows where the resource is located, the server will pass a redirection message to the client. The client then retrieves the resource, while the user remains unaware of the redirection.

Unlike the X Window System, where a remotely-executing process must maintain most of the data structures for a locally-displayed GUI, the WWW uses the reverse model. The web server
provides, via HTTP, the client data required to construct a view of the information (in a device
independent form - HTML), and it is the sole responsibility of the client-side browser to render and
display this data in a suitable form. As a result, not only is HTTP a very lightweight protocol, but
HTTP servers are lightweight applications. This makes HTTP suitable for embedded applications
and purposes.

HTTP has primitive support for content negotiation which currently has found little use. Al-
though HTTP itself is independent of any transport mechanism, the de facto transport for which it
has been implemented is TCP/IP, using the BSD socket interface. To quote from [FGM+97]:

‘HTTP communication usually takes place over TCP/IP connections. The default
port is TCP 80, but other ports can be used. This does not preclude HTTP from being
implemented on top of any other protocol on the Internet, or on other networks. HTTP
only presumes a reliable transport; any protocol that provides such guarantees can be
used; the mapping of the HTTP/1.1 request and response structures onto the transport
data units of the protocol in question is outside the scope of this specification.’

2.1.1 Uniform Resource Identifiers

HTTP addresses resources as Uniform Resource Identifiers (URIs). URIs are either relative to the
last retrieved resource by a client, or absolute (containing a scheme for accessing the resource). URIs
are not limited to HTTP as an access scheme. Other schemes commonly in use are ftp and news,
for the FTP [PR95] and NNTP [KL86] protocols respectively.

In its first working incarnation, HTTP/0.9, a client request was as simple as sending the following
string to the server:

GET /index.html

The server would then reply with the contents of the file index.html, located within the document
root directory.

In more modern versions of HTTP, transactions involve the sending of a request from the client
to the server. The request identifies the particular method to be invoked, a URI to invoke the method
on, and a protocol version number, followed by other information (client information, and possible
body-content). See Section A.1 on page 102 for some example HTTP transactions.

2.1.2 Process an URI request

The requested resource identifier (URI) is dynamically interpreted by the content origin server, ac-
cording to the configuration of the various rules of its resource namespace. As shown in Figure 2.1,
the URI may refer to a local file (or perhaps a cached remote file, if the server implements caching),
2.1. INTRODUCTION

Figure 2.2: Multiple Participants in a HTTP Request-Response Chain

the dynamically generated result of an executable program (termed a CGI script), internally generated information within the server (for example, server administration or status).

Additionally, as illustrated in Figure 2.2, the server may act as an intermediatory, interpreting the request and passing it (plus associated request information) on to another server. If this other server is a HTTP server also, the process is termed proxying [LA94], otherwise it is called gatewaying, or interworking. Proxy servers rewrite HTTP request URIs and request/response meta-information, as necessary. They interwork at the application layer, and are thus suitable candidates to tunnel HTTP requests through different network protocols and stacks.

An arbitrary number of proxy servers may exist in a HTTP request-response chain, some of which may implement caching of resources to improve throughput and reduce network utilisation [LA94, NLA96]. Some proxy servers may also implement gatewaying functionality to other resources not accessible via HTTP (for instance, FTP archives).

They typically present a HTTP view of things to their clients, but act as clients themselves to talk to non-HTTP resources. Additionally, many newer user-agents support other common protocols in addition to HTTP.

Once an HTTP client has made its request, the server will process the request and form a response for the client. Server responses start with a status line, which indicates the server protocol revision, a status number (i.e. a success or error code) and its textual equivalent. This line is followed by server information, (response) entity information, and potentially a MIME-encoded message containing entity body-content. Multipurpose Internet Mail Extensions (MIME) is an Internet
standard which was introduced to handle the transmission of data objects in mail messages [BF93].

2.2 HTTP/1.1 Features

Until HTTP/1.1, each request/response required a separate connection [FFBL96]. As of HTTP/1.1 [FGM+97], features such as persistent connections and pipelining were introduced. Persistent connections avail of the fact that most requests exhibit a spatial locality of reference - commonly, when a HTML page is down-loaded, most of the in-lined images it references are also available from the same server [Spe95, THO96, Mog95].

[FGM+97] describes the benefits of persistent connections:

- by opening and closing fewer TCP connections, CPU time is saved and less memory is used for TCP protocol control blocks;
- HTTP requests and responses can be pipelined on a single connection;
- Network congestion is reduced by having fewer packets caused by TCP three-way handshakes, and by allowing the open TCP connection sufficient time to determine the congestion state of the network;
- HTTP can evolve more gracefully, since errors can be reported without the penalty of closing the TCP connection. Clients using future versions of HTTP might optimistically try a new feature, but if communicating with an older server, retry with old semantics after issuing an error report.

It is recommended that implementations of HTTP/1.1 support persistent connections.

Pipelining involves the sending of multiple requests without waiting for each subsequent response in turn. The server will send responses to the requests in the same order that the requests were received. As can be seen from Figure 2.3(B), the use of pipelining reduces overall delay of waiting first for each response then sending next request, illustrated in Figure 2.3(A). Potentially fewer PDUs containing request data need to be transmitted, and server is not kept waiting on network I/O for each new client request.

Byte ranges were also introduced with HTTP/1.1. They allow clients to selectively retrieve sub-sets of a resource, without down-loading the entire document. This allows querying of information in certain file types (for example, index from Adobe PDF), or efficient down-loading of incomplete entity bodies from cached entities.
2.3 Common Methods supported in HTTP/1.1

Many of the esoteric and infrequently implemented options from HTTP/1.0 were removed from HTTP/1.1. Currently, seven methods are implemented for client-server interaction in HTTP/1.1. The client sends these messages as text strings to the server, followed by a resource identifier. Clients which support a revision of HTTP greater than HTTP/0.9 must send their protocol revision along with the request. These three items are separated by white space, and the method information is terminated by a newline\(^4\). Meta-information describing either the request, the connection, or identifying the client agent then follows. The entire request is terminated with two consecutive newlines.

For examples of simple HTTP client requests, and several server responses, see Section A.1.

The seven supported methods as of HTTP/1.1 are:

- **OPTIONS** - request information about communications options available on request-response chain, as identified by the URI. This does not imply the need for any resource retrieval or action.

- **GET** - retrieve whatever information resource is identified by URI. The semantics of GET change to a conditional get if the request message includes an If-Modified Since, If- Unmodified-Since, If-Match, If-None-Match or If-Range header field. The conditional get

\(^4\)actually a carriage-return character and a line-feed (CRLF as it appears in [FGM+97]).
2.4 Dynamically generated entity bodies (chunked encoding)

HTTP/1.1 supports encoding of message bodies in “chunks” [FGM+97, Won]. Each chunk consists of a chunk size number, and a number of octets. An entity-body may consist of many chunks, each one sequentially appended to the previous one.

The very last chunk in a chain has a size of zero, in this way indicating the chain end. Chunked encoding is useful in the streaming of multimedia data, for example intra-frame based compressed video [ISO93].

2.5 Protocol Extension Protocol for HTTP

The Protocol Extension Protocol (PEP), a WWW Consortium work in progress, is being designed to accommodate dynamic extensions of HTTP clients and servers by software components to address the tension between private agreement and public specification. The draft [NCK97] envisages its use in:

- qualification of normal HTTP transactions;
2.6 Future of Interleaved Message Transports

Despite the introduction of pipelining and persistent connections in HTTP/1.1, message requests are still handled sequentially. A response is fulfilled in its entirety before another response is handled. This has a large impact in the perceived download time to the reader.

The traditional approach to this problem (pioneered by the Netscape Navigator browser) has been to open multiple simultaneous connections to the origin server. However, this results in an overall drop in network throughput on TCP/IP networks due to the adaptive congestion control behaviour.

With the development of chunked encoding, standardisation work is now in progress to allow asynchronous, interleaved data delivery [Get96, Nie96]. This will improve responsiveness of the web by diminishing perceived latency, in much the same way as multi-threaded GUIs present greater perceived interaction to the user.

2.7 Future Development of HTTP

The next generation HTTP protocols are popularly (albeit unofficially) referred to as HTTP-NG. As HTTP matures, its emphasis has moved from hypertext to hyper-media (MIME-encapsulated). Now, with support for interleaving, it is becoming a generic transport protocol [Nie96].

The problems being addressed in the development of HTTP include tackling scalability problems (hence the introduction of pipelining, and persistent connections), latency, disconnected operation support and caching (for mobile users), and caching / replication to reduce network load. Even though it was already designed as a read / write system, HTTP support for distributed authoring is being actively developed, to allow direct publishing to the web. Quality of service and real-time...
issues are becoming increasingly important, as HTTP moves towards generic multimedia transport.

Even the entire web model of client-initiated connections is being examined, with scope for development in multi-casting, server-initiated connections call-backs, and asynchronous client notification of server events.

2.8 Summary

HTTP is a very successful application-level protocol. It allows the transfer of arbitrarily-long, multimedia objects (which are type-identified using MIME). Basic content-negotiation is provided, so that client user-agents can indicate preferences for the MIME types they support.

HTTP currently is a very simple protocol, based around a transaction request-response model. At present, bandwidth usage is very much asymmetrical—the request message is often of the order of 100 times or more smaller than response. Technically it is quite difficult to measure request sizes as current servers do not provide support for this.

However, Section A.1 shows a typical request, whereas Section A.2 shows average response sizes for some local servers. This gives a very rough indication of request/response size differences.

HTTP is evolving under the careful supervision of the WWW Consortium into a transport protocol for generic hyper-media. Asynchronous interleaved message transfer, and chunked encoding techniques provide the development hooks necessary for interactive control over the streaming of multimedia data.

HTTP is the most important multimedia transport protocol at present, and its influence is continuing to grow. In later chapters, HTTP will be reviewed in the context of both TCP/IP and ATM, and for QoS support.
Chapter 3

Transmission Control Protocol/Internet Protocol

3.1 Introduction

TCP/IP is the de facto networking protocol of all computers connected to the Internet. All Internet application protocols (including HTTP) are layered upon the functionality providing by the TCP/IP suite. TCP/IP has always been the networking protocol of choice of Unix vendors, and has recently been adopted by Microsoft as the default routable protocol for Windows NT. For ATM environments to provide ease of migration from (and support to) legacy systems, internetworking with TCP/IP is essential.

For this reason this chapter introduces the TCP/IP family of protocols. Attention is focused mainly on TCP, the Transmission Control Protocol. The internal state machine of TCP will be explained, and the algorithms it uses in adaptive congestion control will be presented.

3.2 Brief Discussion on TCP/IP

Developed under DARPA contracts during the 1970s and early 1980s, the TCP/IP suite of protocols provide reliable, end-to-end, connection-oriented, communications as well as connection-less communications (with non-guaranteed delivery). The peers involved in communication may be separate applications running on a single or multiple computers.

Due to its research environment origins, each phase of standardisation was used “in the field” - as a result, the definite specifications of the Internet Protocols are contained in “Request For Com-
3.2. BRIEF DISCUSSION ON TCP/IP

Figure 3.1: TCP/IP Protocol Suite

ments” documents¹ (RFCs), and de facto Unix implementations—particular the 4.4BSD operating system [MBKQ96].

Figure 3.1 illustrates the TCP/IP suite. The Internet Protocol (IP) layer provides routing and addressing functionality to various transmission layers above it (UDP [Pos80], TCP [Pos81b], T/TCP [Bra94]). The Internet Control Message Protocol (ICMP) [Pos81a] is used by gateways to report datagram processing errors to source hosts—for example, if a network or a host is unreachable. Although ICMP utilises IP functionality, it is actually part of IP, rather than a layer above it. Any element wishing to implement IP must also support ICMP.

The sole concern of the IP layer is with routing packets from source to destination. The format of the IPv4 (i.e. the current IP revision in use as of June 2, 2010) is shown in Figure 3.2. The fields in the header serve as follows:

- Version: indicates the format of the header.
- Internet Header Length (IHL) - length of header in 32 bit words.
- Type of Service: provides indication of quality of service required.
- Total Length: length of entire datagram in octets.
- Identification: used to aid in assembling datagram fragments.
- Flags and Fragment Offset: used to aid in assembling datagram fragments.
- Time to Live: maximum time the datagram is allowed to live in internet system. This field is decrement each hop through a router. When it reaches zero, the datagram is discarded.
- Protocol: indicates the next level protocol used in the data portion.

¹maintained by the Internet Engineering Task Force (IETF)
3.3 Transaction Overhead

This section describes the overhead associated with the completion of a simple transaction (request/response) using both UDP and TCP.

3.3.1 UDP

UDP “provides a procedure for application programs to send messages to other programs with a minimum of protocol mechanism.” [Pos80].

![IPv4 Header](image)
Figure 3.3: UDP Transaction Time-line

Figure 3.3 shows the message sequences of a typical UDP client-server transaction (assuming IP datagrams are less than network Maximum Transmission Unit (MTU)). As UDP is connection-less, there is no need for the establishment of a connection prior to data transfer.

The total transaction time is equal to the Round-Trip Time (RTT) plus Server Processing Time (SPT) (assuming, of course, that there is symmetry of transmission paths). To implement reliable transfers over UDP, an application must maintain certain state information about its transmissions and act accordingly (for example sequence numbers, retransmission timeouts, round-trip time estimates, et cetera).

3.3.2 TCP

The services TCP offers application-layer protocols include:

- explicit and acknowledged connection initiation and termination;
- reliable delivery of data, in-order and without duplication;
- flow control and congestion avoidance;
- out-of-band indication of urgent data (not commonly used).

In implementing these features, TCP needs to maintain connection state information. As a result, TCP communication is connection-oriented.
3.3. TRANSACTION OVERHEAD

Figure 3.4: TCP Transaction Time-line

Figure 3.5 shows the format of the TCP header information, which is additional to the IP header for each TCP datagram.

Figure 3.4 shows the message sequences of a typical TCP client-server transaction. Moving from UDP to TCP implies transaction time grows to 2 times RTT plus SPT. The side which sends the first FIN packet (a packet with the FIN flag set - indicating end of transmission) must remain in a TIME-WAIT state for twice the Maximum Segment Lifetime (MSL).

Figure 3.6 shows the state diagram for TCP connections, as specified by [Pos81b], and augmented by [Bra89b]. By tracing through the message sequence of Figure 3.4, it is possible to deduce the particular corresponding TCP state in Figure 3.6.

---

2Figure 3.6 shows the 2^2 MSL timer starting in the TIME-WAIT state. Some TCP implementations start the timer in FIN-WAIT-2, because certain other TCP connections (incorrectly) fail to send FIN. In such cases, the FIN-WAIT-2 state waits forever[Gre96]. Also, any implementations based on BSD code use an MSL of 30 seconds, as opposed to the 120 seconds recommended in [Bra89b].
3.3. TRANSACTION OVERHEAD

---

**Figure 3.5: TCP Header Format**

---

**Figure 3.6: TCP Connection State Diagram**
### 3.3. TRANSACTION OVERHEAD

![Diagram](image)

**Figure 3.7: Lost ACK with and without TIME-WAIT delay**

#### 3.3.3 The Purpose of the delay in the TCP TIME-WAIT State

Any host which performs an active close on a TCP connection must enter the TIME-WAIT state. This state involves maintaining connection state information for a certain delay after the connection has closed.

HTTP servers perform active closes on connections when requests have been served, or after the request count has reached the maximum permitted value on a persistent connection. As such, HTTP servers can have many connections in the TIME-WAIT state at any particular time, using up system resources. TIME-WAIT is necessary however for the correct operation of TCP.

[Ste96] explains quite well the intention of the delay in the TIME-WAIT state. The enforced delay of 2 * MSL provides two very important functions. It allows full-duplex closing of a connection. It also allows the expiration of old duplicate segments.

Figure 3.7(A) shows a lost ACK packet. When the side performing the passive close times out on waiting for this ACK, it retransmits its FIN packet. Figure 3.7(B) shows what happens with the TIME-WAIT delay in operation. The FIN is received, the timer is restarted, and the ACK is retransmitted. Without the delay, the side performing the active close receives a FIN packet which does not correspond to any of its open connections, so it notices this as an error and sends a RST packet (reset) to its peer, possibly generating an error message at that end.

With TCP, the Time-To-Live (TTL) field in the IP header is set to the current value of the MSL.
3.3. TRANSACTION OVERHEAD

Figure 3.8: TIME−WAIT Old Segment Expiration

Each router must decrement the TTL field by one or by the number of seconds it spends routing the packet, whichever is greater. When the TTL reaches zero, it expires and is discarded. In Figure 3.8, any outstanding packets are guaranteed to have expired at the end of the 2 * MSL delay, at which point new SYN requests can be entertained on the same TCP connection.

3.3.4 Slow Start Flow Control

Many TCP connections will encompass several networks on the route between source and destination nodes. Commonly, each network differs in the bandwidth it offers to others, or in the degree of congestion it is suffering. The slowest network in the chain may be receiving more network traffic than it can handle.

However, if the connection could reach a steady state, new packets would enter the network only when previously sent packets were acknowledged. The number of packets in the network would be constant - spread apart by (at least) the transmission times of the slowest network link.

In addition, if periods of network congestion are experienced, a connection should reduce its transmission rate sufficiently - not just to stabilise the congestion but to actually alleviate it.

The TCP state machine presented in Figure 3.6 is therefore extended by a self-tuning, slow start, flow control algorithm [VJ92].

3\textsuperscript{i.e. the same pair of addresses, where an address is defined by an IP address and socket port number.}
A connection starts with a limit of one outstanding, unacknowledged packet in the network. Each time an acknowledgement (ACK) is received, the limit is increased by one packet. If the ACK also contains a window update, then two packets can be sent in response. This continues until the window is fully open.

During this slow start phase, if each packet is separately acknowledged, the limit doubles during each exchange. This results in an exponential opening of the transmission window. Congestion may cause ACKs to be delayed or coalesced, slowing the window opening slightly.

TCP is designed for operation over unreliable, shared-media networks, without the concept of “Quality of Service” (QoS). The appropriate window size for a particular TCP connection depends not only on the route between source and destination, but also on the competing traffic on each network link. The window size therefore must be dynamic in nature, responding to indications of congestion from the network.

Each connection has in its associated state information an estimate of the usable window size for the path [MBKQ96]. When a packet is dropped (as evidenced by the retransmission timeout), it is due to either network congestion or packet corruption. Consequently, the current window size is potentially too large - thus the window size estimate is set to half the number of outstanding octets.

At the same time, the slow start window size is set to one segment, and the slow start algorithm restarts until the window size is equal to the new estimate of the window size threshold.

After reaching this threshold, the window size increases linearly by one segment for each full window of data transmitted. Thus the increase in size during this phase is linear, while the decrease on experiencing congestion is exponential. This is commonly known as the “TCP throttling” of bandwidth at start-up and during periods of congestion.

### 3.3.5 Fast Retransmission and Recovery

TCP detects that packets have been dropped by a timeout, causing a retransmission to occur. Most TCP receivers respond to an out-of-order segment with a repeated ACK for in-order data. Given sufficient evidence of re-ordering, the receiver can assume that a packet has been lost.

The 4.4BSD [MBKQ96] operating system implements a fast retransmission [Jac88] scheme which is more efficient at handling the case where a single packet has been lost than the self-tuning flow control algorithm. After detecting four identical ACKs, the current connection parameters are stored. A retransmission timeout is then simulated to resend one segment of the oldest data in the send queue, after which the current transmission state is restored.

The window size is divided by two, because the dropped packet is taken as a signal of congestion.
3.4. IPV6

However, because the receipt of ACKs has not stopped, the return into a slow start phase is not required.

An acknowledgement for the missing segment, plus all out-of-order segments queued before the retransmission will then be received and the connection can continue normally—without another slow start. This ability to avoid the slow start phase is known as fast recovery. The way in which these four algorithms (slow start, congestion avoidance, fast retransmit, and fast recovery) inter-operate has recently been described in [Ste97].

3.3.6 Nagle Algorithm

The Nagle algorithm [Nag84] is a TCP scheme for reducing small-packet traffic, first suggested by John Nagle. The Internet Host Requirements RFC [Bra89a] states that hosts must implement the Nagle algorithm.

The minimum header for a TCP packet is 40 bytes (between IP and TCP). If a single byte packet is sent, the overhead is 4000%. The Nagle algorithm unconditionally inhibits the sending of new TCP segments when new outgoing data arrives, if unacknowledged data remains.

The algorithm is both simple, and self-tuning. It allows the first octet output to be sent alone in a packet without delay. Until this packet is acknowledged, however, no further small packets may be sent on the connection.

If enough new information arrives to fill a maximum-sized packet, then another packet is sent. As soon as the outstanding data is acknowledged, the queued input may be sent. Only one small packet is ever allowed to be outstanding on a connection at a particular instant.

As a result, data from small output connections are queued during one RTT. If the RTT time is less than the generating source data rate\(^4\), transmissions are never delayed, and the response time remains low.

3.4 IPv6

IPv6 [DH95] introduces some changes to IP to handle the scalability problems suffered by the current version of IP today (IPv4)—the most predominant of these being the 32-bit addressing scheme. Figure 3.9 shows the structure of the IPv6 header. As can be seen in comparison to Figure 3.2, the overall structure is simpler, and the addresses are greatly expanded.

The “Next Header” field is the offset from the IPv6 header to the start of the next extension header. IPv6 allows option headers to be chained together using this field. The “Hop Limit”

\(^4\)as it would be for the inter-character arrival time, for a remote-terminal application on a LAN.
3.4. IPV6

The salient features of IPv6 are:

- An expended addressing scheme. Node Addresses are now 128-bits wide. The concept of an "anycast" address has been introduced, where a datagram is delivered to one of a group of nodes. It also adds scoping to multi-casting, to improve the efficiency of network utilization.

- Simplification of Header Format. The IPv4 header has been simplified to allow greater throughput in routing of IP packets. Some IPv4 options have been dropped, other have been changed. Most notably, fragmentation of upper-layer datagrams only occurs at source nodes in IPv6.

- Improved Support for IP Extensions and Options

- Labelling of Traffic Flows - permitting "soft" support for quality of service and real-time requirements.

- Extensions for authentication and privacy.

IPv6 allows option header to be chained together in the following order:

- IPv6 Header;

- Hop-by-Hop Options Header;

- Destination Option Header;

performs the same function as the IPv4 “TTL” field.
3.5. SUMMARY

- Routing Header (specifies route to destination, similar to IPv4 source-route option);
- Fragment Header (allows the sending of larger datagrams than path MTU);
- Authentication Header;
- Encapsulating Security Payload;
- Final Destination Options Header;
- Upper-Layer Header (e.g. TCP or UDP).

3.4.1 IPv6 differences to IPv4 from an ATM perspective

IPv6 is designed to ease tension on current Internet address space, by increasing it from 32-bits to 128-bits. It has also been developed to allow for more efficient routing, and fragmentation only occurs at source nodes. These features should allow IPv6 to be carried over ATM more easily than IPv4. For a more thorough discussion of IP over ATM, see Section 6.3.

3.5 Summary

TCP/IP has been used successfully for over twenty years in a large-scale internet environment. As network bandwidth and computer processing speeds increased, TCP has scaled well with increased network traffic. The biggest problem currently facing TCP is the fact that the original addressing scheme is too small for the massive demand at present.

IP provides the core functionality necessary to route datagrams between hosts on an IP-based internet.

TCP/IP provides both a very low-cost interface to basic networking functionality for applications (via UDP), and a reliable communication mechanism for guaranteed delivery via TCP. Both UDP and TCP extend the functionality provided by IP addresses to uniquely identify a specific application instance on the addressed host.

TCP has a small, efficient state machine design—comprising of only eleven states. It performs sophisticated congestion avoidance, and adapts its transmission rate dynamically according to its view of the current state of the network.

Unfortunately, there is a very large gap between the functionality offered by UDP and TCP. For simple request-response applications, the three-way handshake initiated by TCP at the start of a connection may be unnecessary overhead. This is certainly the case for HTTP traffic, and also for RPC-based clients and servers.
Chapter 4

Asynchronous Transfer Mode

4.1 Brief Discussion on ATM

Asynchronous Transfer Mode [IT95], the chosen transport technology for Broadband Integrated Services Digital Network (B-ISDN - see Figure 4.1), is in many ways a break from previous traditions of the telephone industry.

Previously the telephone industry saw communications as essentially point-to-point, connection-oriented services for which they could bill their customers. Computer networks on the other hand were oriented toward group interaction of a shared network. The network was a group resource, upon which sophisticated information systems could be built to assist in distribution of information around the group (recently termed “groupware” or “intranet”).

In recent times, a blurring of distinction has occurred between the boundaries of traditional computer networking and traditional telecommunications. The telecommunications industry is looking to build global-scale information systems to avail of their new digital transmission architectures, whereas computer networks are struggling to integrate new information systems which have temporal and spatial aspects (video, voice, etc.)

ATM allows bandwidth of telecommunication channels (such as SONET / SDH) to be appropriately sub-divided into asynchronous units called cells. Each cell is 53 octets in length, with a 48 octet “payload” and a 5 octet header, as illustrated in Figure 4.2.

Applications which have a temporal dependence (the so-called real-time applications) may be granted a guaranteed fixed amount of bandwidth per second. Applications which are independent of time (as typifies many computer network applications such as file transfer, hyper-linked information systems, network directory services) are not given any such guarantee, but are multiplexed with the
4.1. Brief Discussion on ATM

Figure 4.1: B-ISDN Layer Model

Figure 4.2: ATM Cell Structure

real-time services to make the best possible use of channel bandwidth.

4.1.1 Motivations for a new network transport

Why introduce such a new network transport technology? What are the needs for the features it offers? Advocates for ATM would argue the need to avail of transmission media evolution to provide higher bandwidth, to reduce the number of network technologies currently available, and a future-proof technology.

Year by year, the capacities of current communications networks are strained to the limit. There is an ever-increasing requirement for bandwidth, as computer applications become increasingly network-centric. With CPU and system bus design improvements, the network is now becoming a bottleneck. The advances in transmission technology, the deployment of noise-immune optical
fibre, and developments in photonic switching enable a much greater information transmission capacity. ATM is well positioned to take advantage of such capacity—by using next-hop routing, it allows its switching functionality (cell routing) to be performed in hardware, thus dramatically increasing the potential for scalability.

Currently, there is a large proliferation of communications services—most of them requiring a specific communications architecture, due to their differing requirements. Voice traffic is sensitive to both delay and jitter, but occupies little bandwidth compared to video conferencing. Data transfer is insensitive to delay and jitter, but may require varying amounts of bandwidth. ATM hopes to consolidate the different communications techniques into a single flexible architecture. By doing so, it is hoped to gain stability, a better deterministic understanding of the network, lower training requirements, and huge economies of scale.

Another motivation for the introduction and continued development of ATM is quite practical. Since there seems to be a definite need to replace current networks, as they reach the end of their architectural lives and capacities, there is equally a need to create the “right” technology now, future-proofing it to avoid having to repeat this work later.

The cost of replacing existing networks with a new technology (such as ATM) is quite significant—a potentially large amount of cabling network infrastructure will need to be replaced, and software frameworks and applications may need upgrading or replacing. ATM provides for the upgrading of networks in separate stages, providing a migration path from current network architectures. Various schemes have been proposed for encapsulating existing LAN traffic in ATM (Classical IP [Lau94], LAN Emulation [Com95], Next-Hop Resolution Protocol, Multi-Protocol over ATM), so that existing client-server and distributed applications will continue to operate unchanged.

4.2 Cells, Switching and Virtual Connections.

ATM is a connection-oriented communications technique, which uses fast packet switching based on a finely grained bandwidth unit. An ATM connection is described as consisting of the concatenation of ATM layer links in order to provide an end-to-end transfer capability to connection endpoints. In other words, an end-to-end ATM virtual connection may span many switching nodes.

Connection identifiers are assigned to each link of a connection at setup, and released when no longer needed. Being connection-oriented, ATM requires the various switches spanned on a single link to maintain state information about the link, such as quality-of-service parameters (QoS) and next-hop routing information.

Each ATM cell contains two identifiers which uniquely address its immediate source and desti-
nation at a particular node, based on next-hop routing. The identifiers are the virtual path identifier (VPI) and virtual channel identifier (VCI). Conceptually, the virtual paths and virtual channels specified by the VPIs and VCIIs exhibit a hierarchical relationship - a virtual path may contain many related virtual channels (with similar destinations). Figure 4.3 illustrates the relationship between virtual paths, virtual channels, and the physical transmission medium.

A Virtual Channel Connection (VCC) provides an end-to-end connection at the ATM layer. It is the concatenation of one or more virtual path connections (VPCs). A VPC is the concatenation of physical links where switching occurs based on the VPI field. Cell sequence is preserved within VPCs, but cell integrity is not guaranteed. VCCs (also known as Virtual Circuits (VCs)) are bi-directional.

The two distinct types of VCs are categorised by the method of their creation - Switched Virtual Circuits (SVCs) and Permanent Virtual Circuits (PVCs). SVCs can be created and destroyed on demand by an application, by means of a user-network interface, and network-network interface (most commonly ATM Forum UNI 3.0, UNI 3.1, UNI 4.0 and ITU-T Q.2931). Permanent Virtual Circuits are generally only created and destroyed by the physical intervention of a network administrator or operator—commonly from a serial-terminal or, with more sophisticated switches, using a standard management protocol such as SNMP [SDFC89, CMRW93] or JMAPI [Jav97].

The use of signalling to establish VCs allows clients to perform quality-of-service negotiation with the network.

Figure 4.3: VPs, VCs and the Physical Transmission Medium.
4.3 ATM cell format

Previous telecommunications systems have always dealt with bit-streams. In this respect, ATM marks a significant departure from tradition. The “cell” in ATM is the fundamental unit of bandwidth. It is defined as 53 octets, 5 containing header information and 48 with a data payload[IT95]. Figure 4.4 shows the structure of the ATM cell header. At the ATM layer, only the header information is significant - the payload passes through uninterpreted.

The cell header contains the addressing and control information needed for the cell, and along with the fixed-size, allows implementation of cell-switching in hardware.

GFC - Generic Flow Control - historic reasons
VPI - Virtual Path Identifier - 12 bit at NNI, 8 bit at UNI
VCI - Virtual Channel Identifier
PTI - Payload Type Identifier - 3 bit - used to differentiate between user data and OAM cells.
CLP - Cell Loss Priority - divides VCC traffic into two categories as viewed by its associated QoS
HEC - Header Error Check - 8-bit Cyclic-Redundancy-Checksum on entire ATM header.

The HEC algorithm used in the ATM cell header provides single-bit error correction and multi-bit error detection. The algorithm has two states - a correction state and a detection state. Figure 4.5 shows the HEC algorithm state diagram.
4.4 Quality of Service

Any ATM connection has an Quality of Service (QoS), a guaranteed service contract established with the ATM network. QoS is defined by characteristics such as:

- peak and average cell rates;
- burstiness;
- end to end cell delay;
- cell delay variance (CDV);
- cell loss probability.

The significance of each of these parameters depends on the type of traffic the connection will support. For example, constant bit rate (CBR) traffic will focus heavily on limiting CDV, but will not need the peak cell rate or burstiness parameters.

When an attempt is made to setup an ATM connection, the network will attempt to obtain the requested QoS guarantees from all switching nodes traversed from one end to the other. If this attempt fails, the system will reject the call; at which point, the application can either attempt to renegotiate the quality downwards, or indicate an error to the user.

The task of providing end-to-end QoS is non-trivial [Arm94]. It is difficult to predict the interaction between virtual connections carrying packet and CBR services. Packet-based service traffic distributions are most likely to be statistical rather than deterministic in nature. They therefore introduce random variations in the inter-cell gap and end-to-end delay. It is preferable to buffer data, than to lose it, and the end-to-end delay of packet data is inconsequential. On the other hand, most
4.5 Connection Admission Control and Traffic Policing

Connection Admission Control (CAC) is a vital part of the more general issue of traffic management in ATM. It is responsible for determining whether sufficient resources exist to support a connection request, without interfering in a detrimental fashion with other existing contracted connections. The quality of the CAC algorithm is thus a perceived form of QoS. The problems of deciding whether to accept or reject new connections must be addressed in real time, while simultaneously attempting to maximise the overall network throughput and utilisation.

The preventative congestion control approach to ATM traffic policing involves the use of either deterministic or statistical algorithms for multiplexing traffic onto a single physical link. An example of a multiplexing strategy is the leaky bucket algorithm [IT93b, Tel95], illustrated in Figure 4.6. This algorithm suffers in dealing with different traffic types—for example, real-time isochronous traffic, plus computer data. As mentioned, the data traffic requires large buffers to prevent retransmissions, and the isochronous traffic requires small buffers to meet its real-time requirements. It is possible to use different size buffers for insured rate and excess rate traffic, but this significantly increases algorithm complexity [Arm94].

The alternative to preventative control is reactive control. This involves, for example, the setting of the CLP bit in the ATM header (see Figure 4.4), and possibly an Operations and Maintenance (OAM) “heartbeat pulse”. This approach is taken in Frame Relay [Smi93] with its Forward Explicit Congestion Notification (FECN) and Backward Explicit Congestion Notification (BECN) messages.

4.6 ATM Adaptation Layers

The ATM Adaptation Layers (AALs) [IT93a] provide applications and services access to QoS. An AAL maps services from their native format to fixed-size ATM cells, and back. Different adaptation layers are required for different services. Some adaptation layers (e.g. S-AAL) also provide
additional functionality such as assured delivery.

An AAL is split into two sub-layers, the convergence sub-layer (CS), and the Segmentation and Re-assembly sub-layer (SAR) (see Figure 4.7. The CS is further subdivided into the Service Specific Convergence Sub-layer (SSCS) and the Common Part Convergence Sub-layer (CPCS).

The SSCS is application-specific, whereas the CPCS is common to all applications for a given AAL. The functions provided by the CS are service dependent – they may include message framing, error-detection, et cetera.

The SAR adds headers and trailers to CS data units to form cell payloads. At the destination, the SAR reassembles cells into messages.

The SAR sub-layer is concerned with cells, whereas the CS sub-layer is concerned with messages.

The following AALs have been standardised:

- AAL 1 - supports constant bit rate services;
- AAL 2 - supports variable bit rate services with boundary protection;
- AAL 3/4 - supports data services;
- AAL 5 - supports data services.
4.6. ATM ADAPTATION LAYERS

AAL 1 is designed for CBR, connection-oriented, real-time data streams without error detection, but with sequence protection. These services are not sensitive to data corruption, but are sensitive to delay. AAL 2 is designed to support compressed audio and video streams, with VBR traffic profiles. These services require preservation of message boundaries—in the example of video, to synchronise on the next frame. AAL 2 is considered an unworkable, broken AAL standard [Tan96].

AAL 3/4 and AAL5 are the most important adaptation layers in the context of this thesis. They are both designed to support computer data traffic.

4.6.1 AAL 3/4 Adaptation Layer for Data Services (ITU-T)

AAL 3/4 originally began life as two separate protocols. However, as standardisation work proceeded, it became apparent that there was no real need for two separate protocols, and the two were merged into one adaptation layer.

AAL 3/4 is designed for:

- data sensitive to loss but not delay;
- peer-to-peer operational procedures: assured, non-assured;
- services requiring the following SAR sub-layer functions: SAR, error detection, sequencing, multiplexing (Figure 4.8 shows the AAL 3/4 SAR Structure);
- services requiring the following convergence sub-layer functions: non-assured transfer of data
4.6. ATM ADAPTATION LAYERS

frames of any length between 1 and 65535 octets, abort (Figure 4.9 shows the AAL 3/4 CS Structure).

AAL 3/4 has a high overhead of 4 bytes per SAR-PDU of 48 bytes. 10 bit CRC is used for detecting corrupt segments. A 4 bit sequence number is used for detecting lost and mis-inserted segments.

It can operate in either stream or message mode. In message mode, message boundaries are preserved. In stream mode, the boundaries are not preserved. In addition, reliable and unreliable transport of data is available in each mode.

AAL 3/4 also supports multiplexing, allowing multiple sessions from a single host to travel over the same virtual circuit and be separated at the destination.

4.6.2 AAL 5 Adaptation Layer for Data Services (ATM Forum)

AAL 5 is designed to offer a service with less overhead and better error detection below the CPCS layer than AAL 3/4. The service layer of AAL 5 provides identical services to the CPCS of AAL 3/4, with the exception of support for multiplexing of streams. With respect to AAL 5, if multiplexing is required at the AAL layer, it will occur in the SSCS sub-layer.

Like AAL 3/4, AAL 5 supports both message and stream mode, with reliable and unreliable delivery in either mode. It also supports unicasting and multicasting—with multicasting being incompatible with its reliable delivery mode. Figure 4.10 shows the structure of AAL 5.

AAL 5 was originally accepted by ATM Forum as a more efficient adaptation layer. It was later accepted by ITU-T.
4.6. ATM ADAPTATION LAYERS

Figure 4.9: ATM Adaptation Layer 3/4 CS Structure (ITU-T)

Figure 4.10: ATM Adaptation Layer 5 (ATM Forum)
4.6. ATM ADAPTATION LAYERS

4.6.3 SAAL - the adaptation layer for Signalling

The Signalling Adaptation Layer (SAAL) [IT94] provides a highly reliable transmission service to the signalling layer proper above it. This is quite important as this layer (Q.2931) does not possess any error compensation mechanisms of its own. This is implemented using SAAL’s service-specific convergence sub-layer, the SSCOP protocol (Service-Specific Connection-Oriented Protocol), which is built on top of the CPCS and SAR sub-layers of AAL 3/4 or AAL 5. Figure 4.11 shows the logical structure of the SAAL.

The SAAL offers the following features to the signalling layer [Kya95]:

- sequential continuity - ensures SSCOP PDUs are transferred in unchanged order.
- error correction and repeat transmission - the SSCF issues sequence numbers, which ensure that the loss of PDUs is immediately detected. This error can be corrected by selective requests for repeat transmission.
- error-messages to levels management - any errors occurring are reported to errors management.
- flow control - using flow control information, the receiving station can control the sending station’s data transfer rate.
- keep-alive - if no data is transmitted between two SSCOP nodes, for a relatively long time, POLL PDUs are be sent at regular intervals to indicate that the connection is still required. The maximum permissible time between two POLL PDUs is regulated by the keep-alive timer.
- local data retrieval - the individual SSCOP stations can issue selective requests for repeat transmission of specific lost or unconfirmed PDU sequences.
4.7 STUMBLING BLOCKS IN THE DEPLOYMENT OF ATM

- connection control - the job of connection control is to set up and release SSCOP connections or, if connection problems arise, to renegotiate the transfer parameters and buffer size between sender and receiver (re-synchronisation).

- transmission of SSCOP user data - user data for SSCOP stations can also be transferred via the SSCOP protocol. This allows selection between assured and unassured transfer.

- Detection of errors in SSCOP header.

- Exchange of status information between sender and receiver.

As an AAL 3/4 or AAL 5 CPCS can only perform unassured information transfers, a large part of the SSCOP involves guaranteeing the reliable transfer of SSCOP information. The signalling messages’ requirements are interworked with SSCOP functionality by a Service-Specific Coordination Function (SSCF), which is specific to the respective signalling protocol.

SSCOP sends messages, each of which is assigned a 24-bit sequence number. Messages can be up to 64KB, and will not be fragmented. In-order delivery is assured. Missing messages are retransmitted using a selective repeat protocol.

SSCOP is essentially a sliding window protocol with an interesting strategy for acknowledging transmission. Periodically, the sender polls the receiver to determine what it has received, and discards those messages, simultaneously updating its window.

Table 4.1 lists all the SSCOP PDUs.

4.7 Stumbling blocks in the deployment of ATM

ATM is not without its problems. Perhaps the most significant problem is that of infrastructure. The cost of replacing current wired networks with optical fibre is prohibitive. For this reason, it is expected that ATM will first achieve deployment internally within the telecommunications operators, and also within in-house enterprise work-groups who avail of advanced multimedia services.

In addition, whilst being very simple in concept, in practice ATM is a very complex technology, with many (competing) standards from both ITU-T and ATM Forum. There is the expense of training personnel in the use of ATM technologies and equipment.

Detractors claim that ATM is a broken technology. Connection-oriented networks do not handle connection-less traffic very well. In addition, the 48-byte payload of ATM is seen as an unworkable size. The cell size is a common denominator between throughput, jitter, and other requirements. Different cell sizes were debated, with 32, 48 and 64 byte payloads under discussion. The 48-byte payload was a compromise, motivated by the desire to prevent the need for echo cancellers [Pry95].
### Table 4.1: SSCOP PDUs and their functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
<th>PDU Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Establishment</td>
<td>Request Initialisation</td>
<td>PDU BGN</td>
</tr>
<tr>
<td></td>
<td>Request Acknowledgement</td>
<td>PDU BGAK</td>
</tr>
<tr>
<td>Release</td>
<td>Disconnect Command</td>
<td>PDU END</td>
</tr>
<tr>
<td></td>
<td>Disconnect Acknowledgement</td>
<td>PDU ENDACK</td>
</tr>
<tr>
<td>Re-synchronisation</td>
<td>Re-synchronisation Command</td>
<td>PDU RS</td>
</tr>
<tr>
<td></td>
<td>Re-synchronisation Acknowledgement</td>
<td>PDU RSACK</td>
</tr>
<tr>
<td>Reject</td>
<td>Rejection of Initialisation Request</td>
<td>PRU BGREJ</td>
</tr>
<tr>
<td>Assured Data Transfer</td>
<td>Sequenced Data</td>
<td>PDU SD</td>
</tr>
<tr>
<td></td>
<td>Sequenced Data with Acknowledgement</td>
<td>PDU SDP</td>
</tr>
<tr>
<td></td>
<td>Sending status with</td>
<td>PDU POLL</td>
</tr>
<tr>
<td></td>
<td>Receive Status polling</td>
<td></td>
</tr>
<tr>
<td></td>
<td>Receive Status (solicited)</td>
<td>PDU STAT</td>
</tr>
<tr>
<td></td>
<td>Receive Status (unsolicited)</td>
<td>PDU STAT</td>
</tr>
<tr>
<td>Unassured Data Transfer</td>
<td>Unnumbered data packets</td>
<td>PDU UD</td>
</tr>
<tr>
<td>Management Data Transfer</td>
<td>Unassured transfer of</td>
<td>PDU MD</td>
</tr>
<tr>
<td></td>
<td>management data</td>
<td></td>
</tr>
</tbody>
</table>
With a 53-byte cell, the header information accounts for 10% of total size—a large proportion of wasted bandwidth. For WANs (which is most likely the most important network for ATM technology—that of high speed backbones), the cell size is a poor choice—the most efficient payload size is 60 or 64 bytes [Arm94].

ATM especially does not interwork well with packet-based networks that implement their own form of congestion avoidance [Ste97].

In a review of the events of 1996 in [Wee96], a publication for IT professionals, the “non-arrival of” ATM was described as “failure of the year”. ATM is an extremely sophisticated and complex technology, but as of yet does not have an application which convincingly demonstrates its time has come (much like the WWW did for the Internet).

Competition to ATM has come from traditional shared-media networks, such as Ethernet, and FDDI. Switched Ethernet hubs and sensible use of routers can dramatically increase Ethernet bandwidth from shared 10Mbps to dedicated 10Mbps. Ethernet now offers 100Mbps\(^1\), with gigabit Ethernet being standardised, while FDDI-2 also offers 1Gbps. Isochronous Ethernet offers a stop-gap to ATM, providing traditional asynchronous Ethernet connectivity with a synchronous ISDN-type B channel [fISLANTI95].

### 4.8 Summary

ATM is a connection-oriented transport technology which performs fast switching of fixed-sized cells. It has inherent support for hard (i.e. deterministic, network guaranteed) quality-of-service contracts between end nodes. ATM guarantees that cells are delivered in sequence, but does not guarantee reliable transfer of cells.

ATM tries to be the whole grail of networking. The finely-grained nature of the cell allows bandwidth to be allocated efficiently. The fixed size allows for great scalability potential, as switching can be performed in hardware. The small cell size is also a compromise between network throughput and ensuring services meet certain real-time requirements.

ATM is currently severely lacking in sophisticated broadband applications. Support has been added to ATM to provide legacy system interworking, particularly with computer data networks. This is very important for the future of ATM, as the data networks are where new multimedia applications (using audio and video) are being born.

However, the connection-oriented nature of ATM adversely interacts with data networks. Careful buffer planning in ATM switches is needed to both avoid retransmission “storms” from data

\(^1\)In many instances, using the same cabling, thus providing a very cheap and effective upgrade path
applications, and also satisfy low jitter requirements of real-time services.

The strongest motivation for ATM (and its inherent complexity) is the scalability and complete support for all forms of quality of service and all forms of traffic sources (available bit rate, constant bit-rate, variable bit-rate, unspecified bit-rate etc.) Any application dedicated to selling or encouraging the use of ATM must avail of these features of ATM.
Chapter 5

Related Work

5.1 Introduction

HTTP evolution is closely guarded by the WWW Consortium and is discussed in Section 2.7. This chapter describes research work on the WWW architecture and infrastructure which is being conducted externally to the WWW Consortium.

A review of research into hastening the development of applications over ATM is provided, with particular significance being given to HTTP and the Common Object Request Broker Architecture (CORBA) [OMG96], from the Object Management Group (OMG).

Additionally, work on integrating HTTP and CORBA is examined, and future trends in the development of HTTP are discussed.

5.1.1 Research Summary

The AREQUIPA project extends ATM QoS to TCP/IP applications by the use of a specially modified network stack and the addition of several kernel system calls into the developers APIs.

Robert Leitman, a graduate student at the University of Waterloo, wrote an essay describing his investigations into running HTTP natively over ATM, without an intervening TCP/IP layer.

The ANSA Information Services Framework attempts to undermine HTTP's dominance in the WWW architecture, and replace it with a distributed object solution based on the Internet Interoperability Protocol of CORBA.

JAWS, an experimental web server from Washington University, illustrates how adaptive concurrency, in terms of efficient use of operating system features, and aggressive caching strategies achieve exceptionally high-performance web server designs.
5.2. AREQUIPA

The Laboratoire de Reseaux de Communication\footnote{1} has a project entitled “Web over ATM”.

An extension to the Linux kernel [Var] has been developed. Called Application Requested IP over ATM (AREQUIPA), it provides an immediate solution to the problem of accessing ATM QoS from existing IP-based applications without having to wait for standards from the IETF, the ATM Forum or from the ITU-T. It is based on the assertion that many applications either require now or will require dependable QoS when deployed on the Internet.

Applications running upon existing techniques for providing IP over ATM all lack knowledge of ATM and, as a direct consequence, support for QoS negotiation. The Guaranteed Internet Band-
width (GIB) protocol allows reservation of bandwidth by redirection of traffic to certain routes, but it also is unable to provide isolation of concurrent traffic flows for QoS purposes.

Arequipa is essentially an extension to Classical IP over ATM. It allows applications "establish direct point-to-point, end-to-end ATM connections with given QoS at the link level", whilst using standard TCP/IP and its APIs (e.g. BSD Socket API). It seamlessly coexists with normal use of these network stacks. As a result, applications can benefit in a straightforward way from ATM’s inherent ability to guarantee the QoS of a connection, without requiring a new API toolkit, or network framework or paradigm.

Despite the name, it is not limited to IP and ATM, but theoretically works for other protocols such as IPX, Frame Relay, N-ISDN.

Arequipa is implemented as part of the “ATM on Linux” [Alm97] distribution, and successful trials were carried out in an ATM WAN environment using the ACTS JAMES network. Special system calls allow the dynamic association and dis-association of Arequipa with \texttt{inet} domain\footnote{i.e. TCP/IP. \texttt{inet} is a constant defined in the BSD Socket API[REF]} sockets.

Figure 5.1 shows the simplified workings of the TCP/IP network stack in the Linux kernel [Cox96], modified with Arequipa support [ALO97].

Incoming data is de-multiplexed by the stack and queued on a socket. Outgoing traffic is retrieved from a send queue, multiplexed by the stack and sent to the corresponding network interface.

When using Arequipa for incoming traffic, an application makes a \texttt{arequipa\_expect()} system call to the kernel, indicating it expects to receive traffic from an Arequipa VC. If such traffic arrives, the VC is attached to the socket so that out-bound traffic now uses this VC.

When intending to use Arequipa for outgoing traffic on a socket, the system call \texttt{arequipa\_preset()} to preset the socket to use a direct ATM connection.

The Arequipa work also covered the use of web browsers and servers. Through using the Pragma directive as part of the request/response information, and by associating relevant meta-information with data sources (to aid in determining QoS requirements), web clients are able to initiate connections using usual TCP connections, but switch over to dedicated ATM connections (with hard QoS guarantees) should the requested resource need it. In addition, Arequipa-extended clients and servers inter-operate with traditional clients and servers, without change.

Although very intriguing work, Arequipa does have flaws: the application determines the QoS on the channel using a system call, yet more than just application data is transmitted using this VC. TCP/UDP and IP header information is also transmitted in the data-stream, along with application
data. The effect of this is difficult to predict. In addition, the Arequipa connection must be attached to the socket before establishing the TCP connection, or else it must suffer a potentially low maximum segment size [Bra89a, Atk94].

5.3 Adaptation of HTTP for ATM

Robert Leitman, a graduate student from the University of Waterloo\(^3\), wrote an excellent essay on “the method of integrating HTTP with ATM” in 1995[Lei95].

Given the premise that the WWW and ATM are gaining widespread acceptance, Leitman posed the question as to how HTTP should “operate over ATM networks?”[Lei95].

Leitman identified several problems with the direct integration of HTTP and ATM. The most significant of these were:

- Should HTTP be layered above TCP/IP running on ATM, or on ATM directly with no intermediate layers?
- How should the required connection per request characteristic of HTTP/1.0 be addressed? This is no longer a consideration since the standardisation of HTTP/1.1.
- By what means should ATM end nodes be addressable via URLs?
- Should calling party number be provided in call setup message?

The essay chose to layer HTTP directly upon ATM (using AAL-5) and using the SSCOP to perform selective retransmission. This is the same approach taken in this thesis.

Resolving an ATM address from a URL is an interesting problem, and Leitman provides five possible solutions:

- the modification of URL specification to use ATM addresses as well as IP addresses and hostnames;
- the addition of IP as a new style of ATM addresses;
- the provision of ATM address support in potential URL replacements (which would likely also support persistent naming of resources);
- requiring the use of ATMARP from Classical IP;

---

\(^3\) Computer Science Department, University of Waterloo, Ontario, Canada.

\(^4\) Strictly speaking, Leitman's work describes adapting HTTP to run upon ATM.
5.4. INTER-OPERABILITY BASED ON CORBA

- adding ATM address information to Domain Name Service (DNS) systems.

The provision of calling party number is a consideration to provide similar functionality to TCP/IP implementation, for trace logging, generating statistics and for debugging purposes.

Leitman’s work concentrated on HTTP/1.0 which does not support the persistent connections or the pipelining features of HTTP/1.1. Most significantly, it preceded the introduction of the PEP extension mechanism (see Section 2.5, on page 16) for HTTP.

Leitman has submitted a research proposal August 1996 but as yet (June 2, 2010) does not have any published conclusions.

5.4 Inter-operability based on CORBA

The ANSA Information Services Framework [REM+95] is based on inter-operability between the World Wide Web and Distributed Object Computing (DOC) systems based on CORBA. The primary objective of this work is to bring the added benefits of distributed object technology to the Web, without losing any of the features of the Web that have made it so successful.

ANSAweb maps HTTP/1.0 [FFBL96] methods into object methods, and replaces HTTP with IIOP. At the time of publication of [REM+95], HTTP was on revision 1.0, which did not support persistent-connections or pipelining [FGM+97]. As a result, transmitting requests via IIOP methods would be more efficient. The integration also allows direct integration of other CORBA objects—providing a much more powerful alternative to the use of CGI scripts, with significant potential for maintaining state between transactions using server-side “cookies” et cetera.

Figure 5.2 illustrates the early stages of the work - the development of proxy-gateways capable of interworking HTTP and IIOP. In Figure 5.2(A) HTTP requests pass through two HTTP proxies to reach the origin server. In Figure 5.2(B), requests pass through a proxy which converts the request into an IIOP message. This IIOP message is received by an IIOP-to-HTTP proxy-way, and the last part of the trip is conducted via standard HTTP. Responses to these requests will take the reverse route back to the client.

The final stages of the work involve complete integration of IIOP-based web clients and servers with HTTP-based client and servers, through the use of proxies and gateways. Figure 5.3 illustrates the goal architecture.

Interestingly, it is quite easy to run IIOP and HTTP in one server on a single TCP port (port 80). IIOP sends “GIOP” as the first four bytes of any message [Gro95]. It may even be possible

---

5 In fact, Leitman has left the University of Waterloo for a position of employment with Microsoft.
6 Pipelining is actually a dependency of persistent connection support.
7 Cookies are server-side identifiers uniquely naming a particular client session.
5.4. INTER-OPERABILITY BASED ON CORBA

Figure 5.2: Complementary HTTP-IIOP and IIOP-HTTP Gateways

Figure 5.3: Native HTTP and IIOP servers and clients.
5.5 JAWS

The JAWS web-server [HMS97] was explicitly designed by James C. Hu at Washington University to alleviate the performance bottlenecks identified in existing web servers. By the use of adaptive techniques, such as pre-spawned threading, intelligent caching, and prioritised request processing, JAWS managed to outperform every server in the test suite including some commercial servers (such as the Netscape Enterprise server). The most important conclusions reached by JAWS for attaining high performance are to achieve greatest possible efficiency of concurrency strategy, and also to avoid file system I/O. The technical report describing JAWS is entitled “Principles for Developing and Measuring High-performance Web Servers over ATM”, but sadly this is very misleading. The testing of contemporary servers and indeed the implementation of JAWS itself requires TCP/IP. The motivation in mentioning ATM seems to be to provide a fast network testbed for comparing servers—it is likely that typical ethernet would reach saturation point before truly accurate results could be obtained. None of the research is specific to ATM, but rather to high-speed networks in general. The server would have performed identically (and more efficiently) in a high-speed Ethernet environment. Other than this, it is a fascinating paper.

5.6 Evaluations of Related Work

This chapter presented some of the contemporary work in developing both HTTP as a transport mechanism (including its usurpion by IIOP), and in providing efficient WWW-based application functionality upon ATM networks.

The “Web Over ATM” project developed AREQUIPA, a means of performing trivial modifications to existing TCP/IP client-server source code to enable direct access to ATM QoS Support. The HTTP_ATM server allows either client user agent or server to dynamically negotiate communications link QoS.

The Arequipa approach was not adopted in this thesis because it assumes that access will be available to the source code of the operating system’s networking stack. In addition, the QoS contracts negotiated by Arequipa do not take into account the overhead due to the TCP and IP level framing of data.

ANSA’s integration of HTTP and IIOP is an application-level distributed computing develop-
5.6. EVALUATIONS OF RELATED WORK

ment. It bears no direct relevance to ATM, but does provide a useful insight into the direction distributed object computing is progressing in. Although the CORBA IIOP protocol is TCP/IP specific, the facility is present in the CORBA specification to develop an *Environment Specific Inter-Operability Protocol* (ESIOP) to adapt CORBA communication to a native ATM environment. The discussion of such a protocol is outside the scope of this thesis.

By far the most relevant work to the architecture presented in this thesis is Robert Leitman’s essay on the adaptation of HTTP to ATM. Leitman presents some very interesting and novel ideas, including potential solutions to resolving URLs to ATM node addresses. However, Leitman’s work is restricted to HTTP/1.0. It also does not discuss QoS negotiation, dynamic binding of services and protocols, and HTTP extension.

The JAWS project is interesting insofar as it details the development of a high-performance WWW server, but its relationship to ATM is quite negligible. ATM is used only as a fast networking system, and the results of the JAWS project apply equally to other high-speed networks such as Gigabit Ethernet. One of the aims of **HTTP_ATM** is to utilise native ATM features, such as QoS provision.
Chapter 6

Network Transport

6.1 Introduction

This chapter introduces a variety of transport related issues. HTTP implementation and performance over various transport technologies is discussed, as are emulation techniques to enable IP over ATM. The mismatches between HTTP and TCP, and TCP and ATM are exposed.

6.2 HTTP over TCP/IP

It has been well documented that the designs of HTTP/1.0 and TCP/IP are not completely complementary [Spe95, THO96, Mog95]. With HTTP/1.0, each distinct resource request necessitated the use of a separate TCP connection [FFBL96]. With each connection initiation and destruction comes the overhead of another three-way handshake to synchronise the pre-established connection, and a potential period spent in the TIME-WAIT (see Figure 3.6) queue for the closing server socket. As a result, the potential for requiring multiple connection requests to retrieve a single HTML document aggregates the TCP overhead on the link.

Many HTML documents exhibit a spatial locality of reference—most of the in-lined images they reference are co-located within the document on a single server. To this end, HTTP/1.1 introduced persistent connections, and pipelining of requests. Even still, it is possible for a server to perform an active close on the connection due to an error condition (HTTP protocol error, or internal server error) at some stage during a persistent pipelined request batch. This would require additional connection requests to complete the original request batch.

Similarly, there are always going to be in-lined entities (HTML, images, active content\textsuperscript{1} and byte-
codes) that are not co-located\(^2\) with the master document, and these will require multiple separate connections to retrieve the desired data. Most HTTP requests are very short, and the average HTML response is around 2–3 KBytes\(^3\) [Her96]. Local experiments empirically demonstrated the local average at between 5 and 11 KB (see Section A.2). At these sizes, the TCP three-way handshake overhead (see Section 3.3) is very significant. With very small (\(\ll 0.5\text{KB}\)) requests and small responses (\(\approx 3\text{KB}\)), the cost of establishing a TCP connection becomes prohibitive.

While the current standardised protocol is HTTP/1.1, work continues unabated on adding functionality to improve overall execution and throughput. In particular, a content multiplexing protocol (based on similar ideas from the X Window System) is being developed [Get96]. One of the flaws in the current implementation of pipelining is that requests are services sequentially. By providing asynchronous, parallel servicing of requests, overall server throughput would improve. Many current-generation web-browsing clients (e.g. Netscape Navigator 4.x, Microsoft Internet Explorer 3.x) are aggressive in their multiple connection policies. By providing asynchronous pipelining support, it is hoped to simultaneously improve efficiency, while reducing network and server processing overhead.

### 6.2.1 T/TCP

An interesting solution for the overhead of the three-way handshake has emerged in the form of a modified TCP state machine—*TCP for Transactions* (T/TCP). T/TCP reuses the existing TCP algorithms once a connection is established (i.e. the slow-start, the fast retransmission,...) but bypasses the initial SYN-based handshake. This is achieved by using a process known as TAO, *TCP Accelerated Open* [JBB92, Bra92, Bra94].

T/TCP assigns a unique identifier, called a *connection count* (CC) to each individual connection a host makes. Each host supporting T/TCP maintains the most recently-used CC for each peer host for a period of time\(^4\).

When a server receives a SYN request with a CC value greater than that remembered for the most recent connection from the peer, it is guaranteed to be a new connection request. Thus, the usual three-way handshake can be skipped. This condition is known as the TAO test. When the test fails, a graceful degradation occurs to the appropriate three-way handshake state, thereby ensuring the received SYN is for a new connection request, and not a duplicate from an earlier incarnation.

\(^2\)Perhaps “dis-located”?

\(^3\)for static data, which makes up the majority of resources. Images and byte-codes are likely to be larger in size, whilst CGI executable and other gateway output may exhibit greater latency due to larger server processing times and additional network usage.

\(^4\)typically in a modified routing-table structure.
Interestingly, the time spent in the \texttt{TIME-WAIT} state is reduced from 240 seconds to approximately 12 \cite{Ste96}

The applications of T/TCP extend beyond just the use of HTTP. Many other protocols and services do not benefit from the additional overhead charged by TCP for its features, and instead are implemented over UDP. However, UDP is unreliable, and thus the applications end up with custom ARQ or similar implementations. T/TCP could greatly benefit these applications. In short, potential uses include \texttt{Domain Name Service} (DNS) queries, \texttt{Remote Procedure Calls} (RPCs), HTTP, and use with the \texttt{Simple Mail Transfer Protocol} (SMTP).

T/TCP adds seven additional states to the 11 states shown in Figure 3.6. The new states, however, are all extensions to the existing states.

### 6.2.2 Resource reSerVation Protocol - RSVP

The goal of the \texttt{Resource reSerVation Protocol} (RSVP) \cite{BZB+97} is to provide efficient support for internet applications that require quality-of-service guarantees.

An RSVP session is defined by a destination address and an IP protocol. One RSVP session can carry multiple data-flows, defined by any field in the IP headers. RSVP keeps reservations active as long as necessary. In addition, reservation information is periodically refreshed through retransmission of control messages.
Reservations are receiver-initiated to enable support for large groups, dynamic group membership, and heterogeneous receivers.

The receiver obtains information about the senders through special PATH messages, chooses reservation parameters, and sends a RESV message towards the sender. This reservation state merges with the other receiver’s reservation state at the sender.

In RSVP, reservation specific resources and filtering (deciding which packets may use the reserver resources) are decoupled through the use of flow-specs and filter-specs. This provides support for various different reservation styles.

RSVP routers must intercept and handle PATH and RESV messages, pass new reservation requests on to an admission control function, reserve resources on acceptance of a request, and release them on timeout. It must also send confirmation and error messages.

RSVP’s soft-state QoS is more efficient for large and dynamic groups than the equivalent ATM hard-state functionality [Oec96b].

6.3 TCP/IP over ATM

This section describes the various ad-hoc emulation and interworking schemes to provide TCP/IP and legacy application support over ATM networks.

6.3.1 Classical IP over ATM

The Internet Engineering Task Force (IETF) introduced the simplest approach to providing IP over ATM, Classical IP. Classical IP, ATMARP and InARP are defined in [Lau94].

The goal of Classical IP is to allow compatible and inter-operable implementations for the transmission of IP datagrams and ATM ARP requests and replies over AAL 5.

Classical IP uses the concept of a Logical IP Subnetwork (LIS), which is a collection of hosts in the one IP subnet, all operating on the same ATM network).

In a PVC only environment, IP addresses are mapped to PVCs manually. Each node has a “routing” table which specifies which VC corresponds to which IP address.

For SVCs, some method is necessary to determine the ATM address of a node, give its IP address. To achieve this, Classical IP uses what it calls the Asynchronous Transfer Mode Address Resolution Protocol (ATMARP).

Network nodes register their IP and ATM addresses with an administrative ATMARP server on startup. Each separate administration entity configures its hosts and routers within a closed LIS. Each LIS operates and communicates independently of other LISes on same ATM network.
Hosts in the same LIS communicate directly, and communicate to hosts outside the LIS using standard IP router, in this instance, an ATM endpoint attached to ATM network that is configured as a member of one or more LISes.

The default maximum transmission unit for Classical IP is 9180. IP packets are carried inside AAL 5 protocol data units (PDUs). Each PDU contains a header to identify the protocol, followed AAL 5 trailer with padding, length, and a CRC.

ATMARP is the same protocol as the ARP protocol presented in [Plu82], with extensions needed to support ARP in a unicast server ATM environment. InATMARP, similarly, is the same protocol as InARP [BB92], but applied to ATM networks.

Classical IP’s single subnet operation limits the potential use of ATM in IP networks, because there is no means of switching traffic between IP subnets over an ATM internetwork, other than to use a conventional IP router. Additionally, [Lau94] does not support IP multicasting.

### 6.3.2 LAN Emulation

LAN Emulation (LANE) [Com95] takes a different approach to that of Classical IP, and instead works with MAC layer PDUs. LANE maps ATM addresses to MAC addresses (e.g. ethernet addresses). As such, its operation is independent of the network protocol above it (e.g. IP, IPX). The ATM network appears to legacy LAN devices to function as a connectionless LAN. Figure 6.2 illustrates the LANE protocol architecture.

The LANE ATM network emulates a bridged LAN. A bridge is an intelligent repeater which tries...
to avoid unnecessary forwarding of packets. Like Classical IP, data is transported over AAL 5. The MTU for LAN Emulation is the MTU of the emulated LAN (1500 bytes for ethernet).

The components of a LANE environment are:

- **LAN Emulation Client (LEC)** - The LEC is the entity in end systems which provides a standard LAN interface to the upper layers. It performs data forwarding, address resolution, and other control functions. There is one LEC per host attachment to the ATM network.

- **LAN Emulation Server (LES)** - The LES provides control coordination functionality for the emulated LAN. It registers ATM addresses, and provides a facility for resolving MAC addresses to ATM addresses by replying to them, or forwarding them to other clients and returning the response. One LES exists per *Emulated LAN* (ELAN). An ELAN is a virtual IP network.

- **LAN Emulation Configuration Server (LECS)** - The LECS configures ELANs. It implements the assignment of individual LANE clients to different ELANs, based on its own policies and configuration, and the information provided by the LANE client. One LECS exists per LANE domain.

- **Broadcast and Unknown Server (BUS)** - The BUS emulates LAN broadcasting and multicasting functionality by forwarding packets to all known or specified ATM addresses in the ELAN. The BUS also handles initial unicast frames which are sent by a LEC before the data direct virtual connection to the destination ATM address has been resolved.

*Figure 6.3* illustrates the organisation of these components in two example ELANS.

The advantage of LAN Emulation are that it hides all the properties of ATM to higher layers - all legacy systems work without protocol or software changes. However, this also means that there is not provision for the use of ATM QoS contracts.

### 6.3.3 “Routing over Large Clouds”

Classical IP suffers from the problem that to switch IP packets from one LIS to another, a conventional IP router must be used. The IETF “Routing Over Large Clouds” working group is developing a new protocol to resolve internetworking addresses (IP or otherwise) to next-hop Non-Broadcast Multiple Access (NBMA) [LKPC97] networks, including ATM. The protocol is called the Next Hop Resolution Protocol (NHRP) [LKPC97].

The next hop is either the request destination, or a router at the edge of the NBMA network.
Each LIS is required to have one Next Hop Server (NHS). Hosts in the LIS register with their NHS on initialisation.

To resolve an address outside its own LIS, a host sends the next hop request to its NHS. The NHS forwards requests onwards to other NHSes until they arrive at the correct destination. The destination sends replies through all intermediate NHSes, which cache the replies.

In order to supply the NHSes with the addresses of other NHSes, they must either be manually configured or integrated with IP routers.

### 6.3.4 Multiple Protocols over ATM

The *Multiple Protocols over ATM* (MPOA) [For] project is designed to offer transparent emulation of routed protocols over ATM. It offers a similar service to the network layer that LANE offers to the MAC layer.

MPOA interoperates with LAN Emulation. In the MPOA architecture, hosts and edge devices use LANE to forward packets within bridged ELANs. MPOA servers are used for the routing of packets between different network layer subnets (IP, IPX, etc). Routing protocols are only implemented in MPOA servers.

All connections established by MPOA are end-to-end—there are no intermediate router hops.
The complexity of MPOA may be an issue, especially as much of it arises from support of non-IP protocols [Oec96a].

### 6.4 Deficiencies of TCP/IP from an ATM perspective

Initially, TCP/IP over ATM was a priority to enable existing applications run over ATM networks, while also allowing for the development and co-existence of newer QoS-aware multimedia applications.

Much of the core TCP/IP (IPv4) functionality was developed in the late 1970s and early 1980s, when network hardware was not as reliable as today. With ATM over optical-fibre networks, the bit error rate is likely to be in the region of $10^{-10}$ to $10^{-12}$ [Pry95].

Modern implementations of TCP contain four algorithms that are tightly interdependent, which improve the performance of TCP on Ethernet networks [Ste97]. TCP reproduces much of the congestion control and avoidance present in ATM traffic management, although in an orthogonal and destructive fashion.

The slow setup times of ATM switches impact adversely with short lived transaction based client-server applications.

### 6.4.1 IP Overhead for ATM Networks

Even without running one of the emulation schemes mentioned previously (Classical IP, LANE, MPOA), the associated overhead of running IP over ATM is quite considerable.

The overhead from using SDH or SONET OC-3c framing is approximately $3.70\%$. ATM adds an additional $9.43\%$ to this due to cell header information (5 byte header for each 53 byte cell).

From Figure 4.10, it can be seen that AAL 5 adds 8 bytes of trailer information to each data unit, with between 1 and 47 bytes of padding (since each AAL 5 PDU is limited in size to 65,535 bytes). This averages at $6.41\%$ overhead [Cav94] for AAL 5.

For an MTU of 576 (Internet default), IP accounts for $3.47\%$ of protocol overhead, and TCP takes $3.60\%$ additionally on to this.

The combined percentage bandwidth overhead between ATM and TCP/IP totals $26.61\%$ of an OC-3c connection, before the application can utilise any of it.

Due to the relative inefficiency of ATM for small down-loads, KEEP_ALIVE becomes very useful. This is a known problem, similar to phenomena currently occurring with TCP/IP networks, and most browsers and servers now support KEEP_ALIVE for HTTP over TCP/IP.
6.5 Deficiencies of TCP/IP from an HTTP perspective

TCP is by no means the perfect network protocol for HTTP. Its slow-start behaviour impedes the transaction based nature of HTTP [THO96]. TCP currently provides no facility for reservation of bandwidth or quality of service support for multimedia traffic, although this is being developed [BZB+97].

With the average HTTP transfer size somewhere between 0 and 15KB, TCP’s congestion avoidance algorithms reduce potential throughput. The fact that TCP entities performing an active close must maintain connection state information increases the system resource requirements of heavily loaded HTTP servers. In addition, the Nagle algorithm affects the transmission of small packets on TCP networks. Multiple invocation of the `write()` system call can trigger this algorithm, which may unintentionally hamper performance.

6.6 Using HTTP as access protocol for Broadband resources

The Olivetti Research Lab Project Medusa (see Figure 1.2) introduced the idea of separation of control and data functionality in a broadband multimedia network.

The Olivetti approach is to directly connect multimedia devices to the ATM network, and have them act as data sources and sinks.

All these devices run Olivetti’s custom real-time operating system (“ATMos”), but need to speak a standard protocol to allow them to be accessed from the desktop and from legacy and future equipment. I propose the Hypertext Transfer Protocol as such a protocol. Control connections (through, for example, the invocation of a CORBA method) can trigger an HTTP request/response action.

Figure 6.4 depicts a potential architecture for the control of multimedia data sources and data sinks, based on Project Medusa concepts and the separation of data, control logic, and control GUI.

HTTP responses can be of arbitrary length. HTTP supports MIME typing of responses, and content-type negotiation of the data stream. It is a simple, lightweight stateless protocol. Work in developing chunked-encoding techniques and asynchronous multiplexing of response data generation will enable increased interactivity of control over response streams.

The SSCOP layer provides selective retransmission of PDUs, and as such offers reliable transmission capabilities for a HTTP layer over an AAL 5 connection.

As HTTP moves from a hypertext transport to a generic object transport, its suitability in choreographing multimedia resource access increases.
6.6. USING HTTP AS ACCESS PROTOCOL FOR BROADBAND RESOURCES

Figure 6.4: Proposed Generic Physical Architecture for Control of Medusa

Figure 6.5: HTTP_ATM Architecture
6.7 Summary

This chapter presented a review of some of the layering problems with HTTP upon TCP/IP, upon ATM. In particular, protocol mismatches and duplication of effort dramatically reduces available network bandwidth, and increases required network processing.

The various techniques for providing TCP/IP upon ATM were discussed. TCP for Transactions was introduced as a potential replacement of TCP for use in transaction request-response applications, such as HTTP client-server interaction. The Resource Reservation Protocol was introduced as an attempt to add soft quality of service guarantees to IPv4 and IPv6 traffic flows.

Finally, HTTP was proposed as a generic multimedia access protocol. In the context of the Project Medusa research work, where control of multimedia streams is separated from stream origin and destination, HTTP is a perfect candidate for the coordination of data source and data sink.
Chapter 7

Design and Implementation

7.1 Introduction

This chapter will describe the design and implementation of HTTP_ATM, an experimental server with native support for servicing WWW requests directly over ATM. The server was implemented to investigate and demonstrate that HTTP natively over ATM is a viable proposition, and that simple extensions to HTTP allow access to ATM’s “hard” QoS support.

The design targets for the server will be discussed. The Object Modeling Technique (OMT) [RBP+91] design methodology will be introduced, and the design of the server will be presented using OMT views of the system. The evolution of the design will be discussed, in particular how the various implementation decisions (choice of API, development language, et cetera) had a significant influence on the refinement of the initial design. Deficiencies in the various APIs will be introduced, with potential workarounds compared.

7.2 Design Targets

In designing an experimental HTTP server, with native ATM support, the following goals were identified:

- **Support for legacy systems**
  
  It is very important the server both inter-operate with existing (TCP/IP based) clients, but also allow native ATM clients access to web resources from TCP/IP servers. To this end, proxying of web requests is supported. Figure 7.1 (A) shows proxying of requests from a TCP/IP based client through to an HTTP_ATM server. Figure 7.1 (B) shows an HTTP_ATM client proxied to a
7.2. DESIGN TARGETS

Figure 7.1: HTTP proxying between different network transports

legacy TCP/IP based server. Proxying is already in common use in the WWW, especially to allow access through firewalls and other security mechanisms.

- **Modularity**
  
The server should attain as much separation between the network functionality (termed the network transport), application protocol (HTTP, etc.) and resource (namespace) service as possible. The namespace service is responsible for taking a resource name specified in a HTTP request, and returning the appropriate resource, encoded in a HTTP/MIME response.

Section 2.1 introduces the concept of a server resource namespace. It is depicted in the illustration of Figure 2.1, on page 11. Depending on how a server is configured, resources may be resolved into any one of a number of static or dynamically-generated data sources. This is explained in greater depth in Section 2.1.

Depending on the request, it may resolve to static information from the file system, or dynamically generated information from internal to the server, or external agents and gateways [McC].

- **extensibility**
  
It would be advantageous to allow arbitrary bindings of protocols to network transports, both from a testing point of view and also to achieve greater flexibility in services and features. The server must continue to inter-operate with legacy clients, whilst allowing newer clients avail of the advanced features of ATM, such as QoS negotiation.

Additionally, the server should be able to add and remove support for new network transports and service protocols dynamically, without the need for re-compilation. This in effect will require dynamic loading of transports, server protocols and namespace services.
7.3 OBJECT MODELING TECHNIQUE

- efficiency
  The design of the server must not impede on the overall server efficiency.

7.3 Object Modeling Technique

The Object Modeling Technique is a methodology combining three views of a system:

- The **object model**, which represents structural aspects and relationships between actors in a system;

- The **dynamic model**, representing the temporal, behavioural features of the system;

- The **functional model**, which corresponds to the transformational, data manipulation aspects of the system

It is claimed in [RBP+91] that these three views of a system separate it into orthogonal components, each of which can be individually examined and understood to a large extent. On a review of the OMT methodology, it appeared to support the architectural and modelling constructs required for the HTTP_ATM project.

The object model visual notation from OMT is used in this chapter to introduce the design of the HTTP_ATM server. A description of the various elements in this notation is provided in Section B.2. Some criticisms of perceived flaws in the notation are discussed in Section B.3.

7.4 Initial Design

Figure 7.2 presents the first draft object model for the system. In this design, a HttpDaemon process is responsible for the establishment of various RequestServers (one per network endpoint). The RequestServers listen on their respective Service Access Point (SAP), and respond to incoming connection requests by creating a RequestSlave to handle and process the request. In this way, the RequestServer is free to monitor the SAP for further connection requests. In this context, the term “SAP” is used to represent a single addressable RequestServer.

Although not explicitly obvious from the diagram, it is assumed that the method used to achieve concurrency between master RequestServers and child RequestSlaves is to split the system dynamically into separate processes\(^1\).

In this model, the actual network API is abstracted through a NetTransport interface - a class hierarchy, providing network functionality implementations for both TCP/IP and ATM/AAL5.

\(^1\)This is quite easily achieved in Unix, using the `fork()` system call.
7.4. INITIAL DESIGN

Figure 7.2: Initial Object Model of HTTP_ATM Server
Logging of system messages et cetera is done through the HttpDaemon, which maintains links to various LogObjects to audit itself.

The illustration shows that there is a constraint placed on the RequestServer, RequestSlave, and Request classes – all must aggregate the same NetTransport specialisation. In other words, a RequestServer listening on TCP/IP SAP creates a RequestServer to handle communications on the TCP/IP connection, and responds to an incoming Request to the TCP/IP SAP\(^\text{2}\).

There are a number of easily identified shortcomings in this design. All entities which require logging must somehow communicate with the HttpDaemon, since only it maintains auditing links with various LogObjects. The Request class creates an unnecessary binding between the application protocol used to service the request, and the request itself.

There is significant overhead involved in the creation of a new process, and also in context-switching from one process to another\(^\text{3}\). For a heavily-loaded server, the separate process model offers quite poor performance. [Ste96] presents evidence that it is quite common for WWW traffic to severely stress both a web server architecture and the quality of the underlying operating system network implementation.

There is little flexibility in this design, especially as the two specialised network transports are statically linked with the system.

### 7.5 Dynamic Loading of NetTransports

On reviewing the object model design presented in Figure 7.2, a potential separation became apparent between the underlying network transport technology, the application protocol used to make the transaction between client and server, and the system service used to resolve the request and form a response.

This added much to overall encapsulation and modularity, and hinted at the possibility of even greater system flexibility. Could network transports, protocols and services could be mixed and matched at system startup, via a system configuration file? Could they be dynamically reconfigured during the lifetime of the server?

The development environment, Solaris 2.5.1, provides a solution via its dynamic linking API. Dynamic Linking is the mechanism by which an operating system process dynamically binds at runtime to the appropriate libraries of code it needs for its successful execution. An extension of this is

\(^\text{2}\)Obviously, it does not make sense (nor is it possible) for a RequestServer listening to a TCP/IP SAP to deal with a Request on an ATM SAP.

\(^\text{3}\)Process priorities need updating to decide which process to switch to, virtual memory mappings need to be changed, and process data may need to be paged in from disk.
dynamic loading—providing the ability for the developer to arbitrarily bind with libraries of code at run-time, to interrogate the library for specific functionality and avail of it if found. More detailed information on the Solaris dynamic loading functionality may be found in Section C.5.

The three identified hierarchies which would benefit from dynamic loading are the NetTransport, the application Protocol, and the request name space Service.

The introduction of modularity into web servers was first proposed in [TBW96]. In this instance, it was restricted to resource name space services. Many current web servers offer this facility (Netscape, Microsoft, Apache, NCSA). In addition, an unofficial patch to the source tree of the popular Apache server allows the dynamic loading of service modules, but again this facility is limited to resource name space services.

7.5.1 Dynamic Loading and C++

It had initially been hoped to structure the dynamically loadable modules as illustrated in Figure 7.3. This object model shows a common Module interface which must be implemented by all sub-classes wishing to be dynamically loadable. In addition, all modules implement a static class method

\[ \text{static Module* Module::Clone(void) = 0; } \]

which returns a pointer to an object of the specialised class, created on the heap. It was intended to look-up this method in the class object when it is dynamically loaded, and invoke it to obtain an object pointer. This is quite a clean, succinct way of controlling the class whilst providing dynamic loading and instantiation capability. The use of static member functions to add to class handling functionality, while maintaining encapsulation, is promoted in [Tal94].

However, the dynamic loading facilities of the Solaris development environment support the C language, and C interfaces [KR88]. With this functionality, it is possible to query an object file for a C function, and if the query is successful, invoke the function. However, the choice for the system implementation language was C++ [Poh97, SE90]. C++ is a superset of C, with object-oriented extensions. C++ compilers are almost entirely back-compatible with C code. C++ supports function overloading, whereby a particular function is chosen at compile time based on its name and the types of its parameters. To provide this feature, C++ compilers transparently encoded both the source code function name and parameter types into a new object file function name, a process referred to as name mangling.

---

Solaris is not unique in its support for dynamic loading. Other SVR4-based operating systems support such features, as do Linux and Windows NT. However, dynamic loading is noted to have its origins with Solaris. The Java Programming Language provides a mechanism for the dynamic loading, verification and execution of bytecode during thread execution.
7.5. DYNAMIC LOADING OF NETTRANSports

![Dynamically Loadable Modules Diagram]

Figure 7.3: Dynamically Loadable Modules
7.5. DYNAMIC LOADING OF NETTRANSPORTS

In addition, C++ provides vtabs, virtual tables of function pointers to allow support for polymorphism through base class pointers\(^5\). Pure virtual functions, in C++, are methods without implementation—any class inheriting the base class interface must provide an implementation for these methods for compilation to be successful.

Looking for a C++ function by its source code name in an object file through dynamic loading will fail, since the name is mangled with the parameter list types. Different compilers are encouraged by [SE90] to use different mangling algorithms. The motivation behind this is succinctly explained in this except from [Buc95]:

> ‘Why can’t I link g++-compiled programs against libraries compiled by some other C++ compiler?’

> ‘Some people think that, if only the FSF and Cygnus Support folks would stop being stubborn and mangle names the same way that, say, cfront does, then any g++-compiled program would link successfully against any cfront-compiled library and vice versa. Name mangling is the least of the problems. Compilers differ as to how objects are laid out, how multiple inheritance is implemented, how virtual function calls are handled, and so on, so if the name mangling were made the same, your programs would link against libraries provided from other compilers but then crash when run. For this reason, the ARM\(^6\) encourages compiler writers to make their name mangling different from that of other compilers for the same platform. Incompatible libraries are then detected at link time, rather than at run time.’

While it is possible to de-mangle GNU C++ object files\(^7\), using a mangled name in the source code greatly restricts portability of the system.

In addition, it proved impossible for subclasses to inherit from a common Module base class. Terse, esoteric linking errors occurred during the dynamic execution of the program, with failed implicit symbol lookups in the object files. From the error messages, it was possible to deduce that the problems were due to a lack of support for vtables. Providing trivial implementations\(^8\) for these methods in the direct base class (NetTransport, Protocol or Service—not Module) and removing Module from the class hierarchy solved these problems.

To create the instance of the class, a function with C linkage was used, returning a pointer to an instance of the specialised class.

```c
extern "C"
{
    Protocol* _CreateProtocol(void);
}
```

\(^5\)assuming the specialised class publically inherits the base class
\(^6\)The ARM is [SE90].
\(^7\)using the supplied c++filt utility
\(^8\)i.e. coding {} for method bodies.
The C++ linkage problems fundamentally shifted the design away from the object model presented in Figure 7.3. \cite{RBP+91} encourages independence of design and implementation. However, it was decided that incorporating Figure 7.3 into the overall design would be misleading. The project implementation language was C++, and was not going to change. Consequently, presentation of a module class hierarchy might unduly create incorrect impressions as to how the system functions.

### 7.6 Design Patterns

In the proposed design, certain design patterns were evaluated to determine if they provided clean solutions to certain design problems encountered. This section describes the concept of a design pattern, and discusses three such patterns which initially looked as if they were applicable to the server design.

#### 7.6.1 The “Object” Problem

Designing reusable object-oriented software is a non trivial task. A good design is being pulled in three distinct, orthogonal directions (see Figure 7.4):

- the system should support high re-usability of objects and frameworks (both source and design)
7.6. DESIGN PATTERNS

- it should offer enough flexibility to change (avoid redesign), and accru low software maintenance (and evolution) costs

- it should be specific to the problem at hand

Quite often, the complexity and difficulty in object-oriented development stems from problems in separating design from implementation (separating the software architect from the programmer). This was empirically demonstrated in this project by the problems of dynamic loading presented in Subsection 7.5.1.

How natural is the progression from analysis through to implementation? Should the design be completed first, and then followed by the implementation? Or should it be an iterative process of design and implementation?

Problems with design errors are pushed down into implementation kludges and hacks if the design is incorrect - similarly, implementation issues are forced into non-optimal design structures if too much attention is focused on lower layers at an early stage in a project.

An interesting twist on Heisenberg’s “Uncertainly Principle”, in relation to object-oriented software engineering, was presented in [OM97] - how do you know what to design without first implementing it? Nevertheless, it is difficult to dispute the fact that it helps to have some form of an architecture to guide all coding.

7.6.2 The “Design” Solution

The concept of a Design Pattern is to record and document an elegant solution (design) to specific re-occurring problems in object-oriented software design. They reflect an untold iterative process of redesign and recoding, succinctly capturing design experience in a form readily amenable to reuse.

The motivation of design pattern cookbooks can be summarised as “Get the design RIGHT faster” [GHJV94].

The theory of design patterns originates not in software engineering but in architecture. Christopher Alexander first introduced a language for describing buildings, and how they should be designed in [AIS+77]. This pattern approach to designing, and the subsequent cataloguing of such designs in “cookbooks” has been readily adopted by the object-oriented community in its quest for the holy grail of software re-usability.

The pattern approach to software development allows the development of a vocabulary for expressing software design concepts, and recurring themes and requirements in the software pro-

9 This separation should only be conceptual. “Everybody does everything”, or at least part of everything, seems to be a recommended approach [OM97] to ensure all developers share common understanding and awareness of common problems.
cess. By formally documenting field-proven solutions and their applicability, a body of literature is forming which will resolve common problems and issues through the development life-cycle. The emphasis in the use of design patterns is not so much on software technology itself as it is on the documentation of sound engineering architecture and design in a software context.

The whole pattern concept is perhaps most succinctly explained by Alexander himself in [GHJV94]:

‘Each pattern describes a problem which occurs over and over again in our environment, and then describes the core of the solution to that problem, in such a way that you can use this solution a million times over, without ever doing it the same way twice.’

The standard components of each pattern in a pattern catalogue or cookbook are:

- the pattern name - which is a handle to describe the pattern, the design problem, solution and consequences. The naming of a pattern increases design vocabulary, thereby facilitating the communication of concepts and ideas.
- the problem - describes in detail where to apply the pattern.
- the solution - describes the element which make up the design, and their roles, relationships, responsibilities and collaborations.
- the consequences - explaining the results, trade-offs and side-effects of applying the pattern.

The main categorisations of software engineering design patterns, as presented in [GHJV94], are:

- Creational - the process of object creation;
- Structural - the composition of classes or objects;
- Behavioural - characterising the ways in which classes or objects interact and distribute responsibility.

The patterns considered in the design presented herein are creational patterns.

Figure 7.5 presents a visual representation of the relationship between application frameworks, design patterns, toolkits and classes. A toolkit presents the developer with a particular API. This API may be implemented by or available to the developer through the use of various objects and classes—or alternatively, it may be a purely functional interface. Design Patterns capture elegant solutions to common software problems, through the use of object-oriented techniques.

The distinction between frameworks and patterns is perhaps more difficult to achieve. While a pattern may be said to capture inter-class invariants (such as how classes interact), a framework structures the roles different patterns take in the design, and how they interact with external classes.
7.6. DESIGN PATTERNS

7.6.3 Relevant Design Patterns

This section introduces some of the design patterns considered for the design and implementation of HTTP_ATM.

Abstract Factory

The Abstract Factory pattern (Figure 7.6) provides an interface for “creating families of related or dependent objects without specifying their concrete classes” [GHJV94]. Its uses include:

- configuring a system with one of a set of product families;
- ensuring a system is independent of how its products are created and implemented;
- ensuring the invariant that a family of related product objects has been designed to work together;
- providing access to interfaces of a class library of objects, not to the implementations.

In the initial design stages, the Abstract Factory pattern was used to enforce a binding between application protocol and network transport - for example, between HTTP/1.1 and TcpIpTransport, or HTTP_ATM and AtmTransport. Figure 7.6 depicts this hard binding feature.

However, as the design evolved, this architecturally structured binding proved somewhat restrictive. A dummy transport was developed (StdIoTransport) which used standard input and output
to test the application protocol operation, but the Abstract Factory structure would prohibit its use. It was decided to remove the structural binding, and to re-implement this instead logically inside the `RequestServer` class.

Prototype

The `Prototype` pattern (Figure 7.7) allows the specification of the class of object to create using a prototypical instance, and creates new objects of this class by copying the prototype. Its uses include:

- instantiation of classes which are only specified at run-time, for example, by dynamic loading
- ensuring a system is independent of how its products are created and implemented (similar to the abstract factory)
- when instances of a particular class can have one of only a few specific combinations of state
- to avoid building factory class hierarchies which mirror that of the product class hierarchies.

With the Prototype pattern, once a client (software program using the interface) has an instance of the prototype, it creates new instances by cloning the existing instance (by means of a `clone()` method – rather like creating new processes in Unix with the `fork()` system call).
The Prototype pattern is quite useful in allowing objects to be dynamically loaded and instantiated. In designing this server, there is a need for this facility. On first impressions, the Prototype seemed to fit the bill. However, retrospective examination proved that implementing dynamic loading from C++ member functions is inherently non-portable\textsuperscript{10}. Languages like Java, where the dynamic loading API is object-aware, or Objective-C, where classes themselves are objects, would not suffer from the same restrictions. In addition to this, there isn’t any need to clone instantiated objects within the server.

Instead, a design similar to the Prototype pattern is used in the implementation. Each module when compiled to a binary object contains a function (of C linkage) which will create an instance of the class on request. This function is located by the dynamic loading facilities and used to dynamically instantiate classes at run-time. Although it technically is not a class interface from the implementation language’s point of view, it is shown as a static class member in the object model (Figure 7.9).

Currently, the interface to create NetTransport modules does not allow the passing of parameter values to the class constructor. While this behaviour may at some stage be useful (for example, to enhance server functionality by providing virtual hosting facilities\textsuperscript{11}), it is not needed in the current single server per transport design.

To facilitate this further development, the logical functionality for module loading and instantiation has been separated for the Protocol, NetTransport and Service modules inside the HttpDaemon class. In comparison, Figure 7.3 shows a single way of instantiating all modules, inher-

\textsuperscript{10}Due to differences in compiler name-mangling algorithms, as discussed in Subsection 7.5.1

\textsuperscript{11}Virtual Hosting is the feature where a single server responds to requests for multiple virtual websites, based on the SAP address the request came in on and also the hostname by which the server was accessed.
7.7. Refining the Object Model

The final object model design is presented in Figure 7.9. From this model, it is possible to deduce that the server consists of several threads, dynamically created during its lifetime. It is the responsibility of the HttpDaemon thread to transform the server into a Unix “daemon”. A daemon is a program

Figure 7.8: Singleton Pattern

ital from the abstract base class, which is more restrictive.

Singleton

The Singleton pattern (Figure 7.8) ensures that only one instance of a class is ever created, and provides a global point to access to that instance. It is a very simple pattern, yet is used to great extent within the server to provide access to the server logger functionality.

Figure 7.9 shows that most classes audit themselves using the logger functionality. The LogObject singleton allows multiple threads and contexts of execution make log entries without worrying about conflicting with one another. The class has a private constructor, so that it cannot be constructed directly in the text segment of the executable or on the stack—it must be constructed on the heap. A static class member function allows access to the LogObject instance, creating it if it does not already exist.

The LogObject constructor automatically registers another static class method to destruct the logger correctly when the server terminates. Using the C library atexit() function, a static member function is registered to release object resources. The LogObject destructor is null, since many apparent instances may exist at the one time, but the singleton only gets “destructed” when the system shuts down. This method ensures that the LogObject singleton is properly destructed, even if it has been dynamically allocated on the heap—it is possible, because there is always only one real instance to destruct.

7.7 Refining the Object Model

The final object model design is presented in Figure 7.9. From this model, it is possible to deduce that the server consists of several threads, dynamically created during its lifetime. It is the responsibility of the HttpDaemon thread to transform the server into a Unix “daemon”. A daemon is a program

Figure 7.8: Singleton Pattern

return uniqueInstance
Singleton
static uniqueInstance
static Instance()
SingletonOperation()
GetSingletonData()
singletonData

Figure 7.8: Singleton Pattern
7.7. REFINING THE OBJECT MODEL

(usually long-lived) that automatically performs tasks transparently to system users [Len88]. It then creates `RequestServers` as instructed. During the steady state operation of the server, a special purpose thread exists with the sole purpose of handling asynchronous signals from the operating system. For each configured network transport and protocol pair \(^{12}\), a `RequestServer` object is created to bind (“tie”) them together to run in a `Thread`. The `RequestServer` object will listen for incoming connection requests on the specified SAP\(^{13}\), passing them on to dynamically created `RequestSlave` objects.

In the design, slave threads are created as needed, with an unbounded limit to the potential number of threads running at any particular moment in time. For heavily loaded situations, this model may not offer the best utilisation of system resources. Alternative scenarios to this include a bound limit of pre-allocated threads, or a producer-consumer thread relationship [Lew96].

### 7.7.1 Threading Model

To evaluate each, it is necessary to put the relevant scenarios in context by first presenting `NetTransport`, the class hierarchy for dealing with network I/O.

Within the abstraction of network functionality provided by the `NetTransport` interface, the stages involved in dealing with connections requests are:

1. Create a network service access point.
2. Bind an address to it. In terms of the `AtmTransport`, this related to an ATM address and a (VPI,VCI) pair. For the `TcpIpTransport`, it corresponds to an IP address and an IP port number.
3. Actively listen for new connection requests.
4. Accept new connection requests, handle them and close the connection.

It is desirable to be able to simultaneously handle processing of requests while simultaneously accepting new requests. Indeed, this is how the current design works.

The pre-allocated server strategy works quite well with the functionality provided by the TCP/IP BSD Socket API (see Section C.4, on page 116). Using the BSD API, A server process splits (“forks”) into a number of identical servers, all of which block waiting to accept a new connection request from a particular SAP (socket). The operating system transparently ensures that only one process

---

\(^{12}\)specified in a configuration file.

\(^{13}\)again, from the configuration file.
will succeed in accepting the connection, per new connection request. Once a new connection has been established, a new SAP is automatically generated for the lifetime of the request, while the original SAP is still available for new incoming requests.

Implementing this from multiple threads within one process produced quite satisfactory results under Solaris.

The producer-consumer strategy works well with the generic NetTransport interface. With this strategy, work accepted by a master server is queued for subsequent processing by slaver servers. The queue may be fixed in size, or may dynamically grow and shrink in a bounded fashion—having an unbounded queue leads to the same problems as unlimited dynamic slave creation, with all the added complexity of the producer-consumer scenario. The queue in this instance may need to be protected from multiple accesses by the use of mutexes and critical sections.

Perhaps the best approach is a variant on the pre-allocated server scenario. Slave threads are initially created up to some low water-mark level. Depending on server load, this number may dynamically increase up to a higher bounded level. When the load on the server subsequently decreases, threads which have been idle for some time may commit suicide as long as the total number of threads is greater than the low water-mark. This should provide low latency for request response, but also provide scalability during periods of high load.

The ATM API [FOR94] upon which the AtmTransport driver was built lacked significant functionality necessary to implement the pre-allocated server scenario. The use of the producer-consumer technique was considered excessively complex for the requirements of the experiment, but could be introduced should the server achieve real-world deployment. The producer-consumer scenario would require the development of a dynamic queue, protected from multiple accesses via mutexes. For the purposes of this experiment, slave threads are created as necessary to handle requests.

Within the BSD Socket API, the abstraction of a network end point is the socket. A socket is a file descriptor maintained by the kernel for a particular process. A server socket is a valid file descriptor for read and write operations only after a complete connection has been established (see Figure 3.6, on page 24). However, within the API, the act of accepting the connection implicitly issues a new socket file descriptor for the connection, allowing the existing server socket descriptor to continue to be used to reference the service access point (so that new connection attempts can be accepted). The ATM API used did not provide such functionality, and so the AtmTransport class

---

14 The operating system guarantees that only one process of a group of processes blocked in an accept() call on a socket will be awoken to handle a single new connection request.

15 The accept() system call is multi-thread safe, according to the Solaris manual pages.

16 The RequestServer and RequestSlave classes encapsulate all the logic which would need to be changed.
wrapped the API in such a manner to provide the appearance of this functionality. This wrapping will be further described when discussing the FORE Systems ATM API (Subsection 7.7.4).

### 7.7.2 Threads

The design presented in Figure 7.9 shows the use of multi-threading. A thread class is associated with a virtual base class, the Runnable interface. In effect, the thread “runs” the runnable object. Instances of classes which implement this interface are runnable themselves. The HttpDaemon, RequestServer and RequestSlave classes all implement (inherit) the Runnable interface.

When a network system calls block (whether due to polling for a connection request or data, or waiting for device I/O to complete), a single-thread process cannot service new requests despite the fact that its quantum may not have been completely used. Using multiple contexts of execution within a single process can potentially boost performance on a heavily I/O bound process, whilst presenting minimal load to the system [Ous96]. The benefits of threads also extend to permit potentially greater performance from the fully symmetric use of multiple processors for computation and I/O, greater interactive responsiveness for GUIs\(^\text{17}\), and potentially cleaner, simpler structure for complex server applications [Mic96].

On the other hand, the use of threads does pose one significant disadvantage compared to the use of multiple processes. With multiple processes, assuming that each process has a finite life (in terms of a number of requests serviced), memory leaks due to mismanaged dynamic allocation of the heap are controlled. With multiple threads, even if the misbehaving thread dies, the address space is per process, not per thread - and so the leak is not controlled.

The thread API used was the native Solaris threads and light-weight process (LWP) API [Mic96, PKB+91]. However, the Thread class encapsulates all the use of this API, and it should be straightforward to port this to use another package such as the POSIX Threads API [IEE96], provided that the threads are multi-tasked in kernel space and not user space (otherwise, any previous gained efficiency is now lost). More information on the thread API can be found in Section C.2, on page 112.

Initially, when the HttpDaemon’s tasks of parsing the configuration file, transforming the process into a daemon, and instantiating the various RequestServers were complete, it was intended that the thread running in the HttpDaemon context would “mutate” into a general purpose signal-handling thread for the system.

\(^{17}\)For example, the BeOS operating system [Be97].
7.7. REFINING THE OBJECT MODEL

Figure 7.9: Final Object Model of HTTP_ATM Server
7.7.3 Unix Signals

A signal in Unix is a traditional mechanism for asynchronous notification of a particular process or system event [Gra97]. Using standard terminology, a signal is said to be sent to a process when the event associated with the signal occurs. Examples of signals are process termination, suspension, floating point exceptions, et cetera.

Processes can arrange to respond to the arrival of certain signals, by creating an association between the signal and a special code routine called a signal handler. With multiple threads per process, the signal is deemed handled when one of the threads receives the signal and takes appropriate action. To ensure that signals do not arrive to threads in a haphazard manner, threads can arrange to block notification of signals, by specifying a signal mask. A special signal handling thread is then necessary, to receive signals and execute the correct response.

The original sequence of events in changing the HttpDaemon into the signal handler was as follows:

- The HttpDaemon would mask out all signals while it created the relevant RequestServer threads. The server threads inherit the mask of their creator, and thus will refuse to handle all signals also.

- Once the HttpDaemon had created all the servers, its work was completed. Now, it would unmask all signals, and block waiting to handle an incoming signal notification.

When this code to perform the transformation was implemented, it became obvious that a subtle race condition seemed to exist. Threads created by the HttpDaemon thread (before it started catching signals) seemed to be the ultimate destination of signals, not the HttpDaemon thread itself. If a thread has a signal masked out, and a signal arrives for the thread, the signal is queued on the thread until such time as the thread unmask the signal.

The solution is to ensure that a special purpose signal handler thread is in place before creating any worker threads. Section C.3 lists skeleton C++ code used to create the signal handler thread.

Having been made redundant after creating all the server threads, the HttpDaemon thread now waits for 5 seconds, before committing suicide.

7.7.4 FORE Systems ATM API

The ATM API available, the FORE Systems ATM API [FOR94], works differently than that of the BSD Socket API for TCP/IP. With the BSD API, once an incoming connection request is accepted, a
new descriptor for the connection is issued, and the original descriptor is still available to block for new connection attempts.

With the FORE API, however, once the connection is accepted, the SAP is used for the entire duration of the call, and must be reinitialised after the call is released.

To simplify the implementation of the server using the NetTransport interface, it was decided to emulate the BSD Sockets functionality in the AtmTransport class.

When the client connects to the server via an AtmTransport connection, the server initialises a new private SAP for the conversation, and passes this address to the client. It then creates a new context-of-execution (COE) (in the implementation, this is a new thread) to watch for incoming calls on this SAP, while it releases the original call, reinitialises the original SAP and continues listening for new requests. This is shown in Figure 7.10(A).

The new COE waits for a certain length of time for the client to re-connect on the new SAP. If it does so, a private conversation can now take place and the clients requests will be serviced. Otherwise, the context-of-execution will destroy the SAP and die. This is illustrated in Figure 7.10(B) and Figure 7.10(C), respectively.

The difficulty in implementing this was due to the fact that the FORE API did not provide a non-blocking function or function which times out to poll for new connection attempts. This is important, because if the client COE dies after the new SAP is created, a server COE is now blocked for incoming calls on this SAP. This COE will listen for the lifetime of the server, and as a result system resources may slowly leak away.

Two possible design/implementation solutions were considered to solve this. One involved a slight alteration of the object model. The other involved adding additional logic to the AtmTransport module itself.

The first solution (the most immediately obvious) was to implement a timeout around listening on the new SAP. The only way to implement timers in Unix is through the standard signal() system call\textsuperscript{18}. To make matters worse, signals and alarms work most effectively at the granularity of a process—they were not designed to operate at such finely-grained a level as the thread.

Signals, whilst incredibly useful, are a poor form of error handling, as they do not allow additional user-defined information to be presented along with the arrival of the signal. Perhaps more importantly, they are delivered asynchronously, which tends to generate awkward convolutions within source code. These are not hugely significant disadvantages within single-threaded, multi-process applications—from which signals originated. However, it can require design alterations and implementation gymnastics to code in a multi-threaded system, as threads and signals have to be

\textsuperscript{18}More specifically, handling the SIG_ALARM signal.
correctly matched\textsuperscript{19}.

For the first solution, the Thread class would be required to maintain a static (i.e. class scope) queue of threads. Before the new server COE blocks on the new SAP, it sets an alarm clock to go off after a specific period of grace. This is achieved using standard Unix C-library `signal()` functionality. When the timeout expires, the process is delivered a `SIG_ALARM` signal asynchronously. The special purpose signal-handling thread will receive this signal, and kills the first thread in listening queue.

This requires a new specific-case association in the object model between the `AtmTransport` module and the Thread class, confusing an otherwise clean design. This new association is shown in the partial object model of Figure 7.11.

The second solution takes the approach that the lack of functionality in the ATM API should not corrupt design with superfluous associations – especially since it is to handle just one particular instance of a `NetTransport` subclass. It makes for a better design to localise the solution to within the `AtmTransport` module.

Once the new COE is created to listen on the private SAP, a separate additional COE is established in which a timer runs (see Figure 7.12). Both COEs have registered the thread identifier of the other in their internal states. If the timer expires before a client connection request is accepted, the timer monitor thread kills the private server thread, and then commits suicide. If a client request is accepted before the timer expires, the server thread kills the monitor.

This design is much neater than that proposed in the first solution. It involves only altering the local module wherein the lack of API functionality caused the problem initially. However, there may be scheduling problems due to the dynamic priorities given to threads in Solaris. The potential is there for race conditions to occur, with a client connection being accepted or pending acceptance. Scheduling the timer thread before the server thread in this instance may cause the timer to prematurely kill the server. In addition, the complexity of the `AtmTransport` class implementation is significantly increased—it now has to deal with the Solaris thread API. Furthermore, the encapsulation of the overall design is impaired somewhat, since this functionality is otherwise completely contained within the interface of the Thread class.

The use of signals is a kernel level construct, and thus is outside the bounds of scheduling phenomena. Even still, it is conceivable that similar situations would occur, where timers will expire before the server thread manages to deregister the alarm. Both approaches are essentially kludges to missing API functionality, and as such it is desirable to minimise their impact on system design. It is quite possible that a future firmware release from FORE will provide the necessary functionality.

\textsuperscript{19}A better, cleaner solution is the use of synchronously generated exceptions, as generated by Windows NT.
at a kernel level to circumvent these application level contortions. As such, the use of a separate monitor thread per `AtmTransport` connection is deemed the most appropriate solution.

### 7.7.5 Service module

The design presented in Figure 7.9 allows various services to be installed in the `HTTP_ATM` server resource namespace.

Currently, the only implemented service is the file service. Achieving the highest possible performance was not a design criterion for the `HTTP_ATM` server. However, it has been demonstrated in external work that over 80% of web server delay and latency is due to I/O [HMS97], particularly file I/O.

To minimise this, the file service avails of memory mapped I/O. Instead of performing multiple user/kernel crossings due to `read()` and `write()` system calls, the file is mapped directly into memory. This allows one `write()` call to send the entire file.

In addition, a hash table of file inodes and `mmap()`ing is held so that a second thread accessing a previously mapped file is spared the cost of opening and mapping the file. Access to this table is mutex-protected, such that a thread can atomically determine whether a file is mapped into memory already and avoid the system calls if possible.

Figure 7.13(A) shows the traditional file I/O approach. Both the `read()` and `write()` system calls may need to be repeated multiple times, depending on the size of the buffers. In addition, multiple user/kernel space crossings may be necessary for each of these system calls.

Figure 7.13(B) shows the memory-mapped I/O strategy. The file is mapped into memory.
7.7. REFINING THE OBJECT MODEL

Figure 7.11: Partial Object Model, showing AtmTransport / Thread Association

Figure 7.12: Monitor thread watching AtmTransport connection.
7.8. **PROVISION OF QOS SUPPORT**

Since the process address space is shared between all threads, a second thread accessing the same file may be spared the cost of performing system calls if it notices the open file in a special inode/file cache. The file inode uniquely identifies the file (as opposed to file name, of which a Unix file can have many). As a result, the file inode is used as the key into the hash table cache. When the write() system is called, the file data is demand-paged into memory and written to the output file descriptor in one user/kernel crossing. The use of this very large write also minimises the potential influence of the Nagle algorithm, discussed in Subsection 3.3.6 (see page 28).

### 7.8 Provision of QoS Support

One of the design goals of HTTP_ATM is to provide access to native ATM facilities—in particular, the Quality-of-Service (QoS) feature. Depending on the medium of a requested resource, it will exhibit different characteristics which identify the most effective means of transmitting its data on the network.

Two levels of associating this information with resources have been identified: by resource type, and by resource instance.

Association by resource type allows for global bindings of QoS information to resources. For example, all resources ending in the extension `.gif` could be transferred as ABR traffic, since they are still images are thus have no real-time requirements. Similarly, all resources ending in `.mpg1` may be sent in a CBR or VBR stream of around 3Mb/s.

With each resource instance, it may be desirable to override the default QoS parameters for that

---

20Actually, the extension is commonly used for the CompuServe Graphics Interchange File format.
21Assuming that `.mpg1` in this instance identifies MPEG-1 compressed video.
7.8. PROVISION OF QOS SUPPORT

type with values specific to the resource. The association by resource instance allows support for this feature.

It is the responsibility of each Service instance mapped into a RequestServer to scan its resource namespace on startup, in order to bind arbitrary meta information with each resource. This information is held as a hash table inside the Service, with the resource identifier acting as the key to the meta-information.

However, the meta information is not interpreted by the Service module, but by the Request-Slave, Protocol and NetTransport modules as necessary.

Currently, the Filesystem service maintains resource instance-specific meta information in a file called .metainfo. Each directory in a directory hierarchy may contain a .metainfo file describing the resources available in that directory. The ftw() library call is used to traverse the hierarchy of directories to read in meta information. If a resource is not explicitly identified in a .metainfo file, its global request type QoS values will be used. If lookup fails for the QoS values of the type, then a default of ABR without QoS will be used.

The meta information file format used by the Filesystem module consists of a number of newline separated entries, with each entry representing a resource. The file is in flat-text format. Entries currently contain a resource name, MIME type information and QoS parameters. The resource name is used by the Service module as the key into the associative array of resource meta information. Everything which follows the resource name is the meta information for that resource.

The MIME type information is used by HTTP/1.1 and HTTP_ATM protocol modules. The QoS parameters are used by the NetTransport and HTTP_ATM protocols.

7.8.1 Negotiation of QoS

Negotiation of QoS may be done by the use of PEP extensions to the namespace (discussed in Section 2.5), or by HTTP_ATM specific header fields.

The following is an example of HTTP_ATM specific header information to provide QoS support:

GET /someResource.ext HTTP_ATM/1.1
Pragma: QoS-negotiation = "FORE; 0, 0, 128, 64, 1, 2"

The use of the Pragma directive is discouraged in [FGM+97]. As such, the more correct method of negotiation is the use of PEP. PEP maps an extension into the server name space. To access a resource through that extension, the extension name is prepended to the resource identifier. For example, consider an extension name textttQoSExt which allows QoS negotiation. The header information takes the form:
7.8. PROVISION OF QOS SUPPORT

GET /QoSExt/someResource.ext HTTP/1.1
PEP: {{map "http://serverHost/QoSExt" QoS}
    {params "FORE; 0, 0, 128, 64, 1, 2" } }

Notice that PEP uses HTTP/1.1 as the protocol version, since PEP is designed as an extension protocol for standard HTTP.

The negotiation actually happens at the transport layer, but the information is carried by the HTTP_ATM layer for logging purposes (informational logging and error recording).

If the extension is not supported (for example, for some reason a client tries this negotiation with a HTTP/1.1 protocol module), the server returns following response status code:

420 Bad Extensions

In this case, the response body will inform the client user (via HTML) that the particular SAP connection does not support QoS negotiation.

If the renegotiation is unsuccessful, then the mapping is considered to have failed, and so the following status code is returned:

421 Bad Mapping

In the event of the renegotiation failing, the user agent should automatically retry for the resource using ABR—the entire negotiation event should be transparent to the end-user.

Status codes 420 and 421 are defined in PEP [NCK97]. As such, they do not appear in Table A.2, on page 104.

7.8.2 Active Party

In the proposed design, QoS negotiation may be client-initiated or server-initiated. The client may specify QoS at connection-startup. This allows the client to give the server a better indication of the available bandwidth to it, and consequently allows the server to make better choices in content type-negotiation, as described in [FGM+97].

In server-initiated negotiation, the server performs a lookup of the URI in the Service hash table to discover QoS and MIME information. It then attempts to renegotiate the QoS service contract at the ATM layer and informs the client in the HTTP response information of the result of this renegotiation. If the negotiation fails, the server reverts to traditional best effort communication (ABR).

The steps involved in server-initiated negotiation are:
1. The client makes a request on a SAP which has the `HTTP_ATM` protocol bound to it, along with the `AtmTransport` net transport.

2. The Server looks up the request in the meta information hash.

3. `NetTransport` attempts link QoS renegotiation, based on meta information obtained.

4. Inform the client of the negotiation outcome in the response header.

Unfortunately, there is currently no support for the renegotiation of QoS in the FORE Systems ATM API. This functionality is simulated at the moment by passing HTTP meta-info around in requests, and returning an outcome value in the response header. No actual renegotiation takes place at an ATM level, though, and the outcome value indicates that the renegotiation is always successful.

### 7.9 Proxy Support

By implementing proxying functionality in `HTTP_ATM`, ATM-based clients are allowed access to legacy (TCP/IP based) servers, and similarly to allow legacy clients access to `HTTP_ATM` servers. Proxying is a part of the HTTP/1.1 standard. Essentially the client passes a fully qualified (i.e. it contains the hostname) URI to the proxy, which acts on behalf of the client to the destination server. Proxying through non-QoS aware servers is best effort only.

### 7.10 Implementation and Results

The `HTTP_ATM` server was implemented using the C++ language, in a Sun Solaris (a Unix SVR4 derivative) environment. The FORE Systems ATM API was used to provide ATM functionality. The Sun two-tier multi-threading API was used to implement server concurrency, and the Sun dynamic loading and binding API was used to provide modularity and dynamic extensibility. These interfaces are described in more detail in Appendix C.

### 7.10.1 Design Features and Development Issues

The `HTTP_ATM` server exhibits some novel design characteristics. The use of dynamic binding of code segments enables some interesting flexibility both in terms of developing the server but also for its actual deployment:
7.10. IMPLEMENTATION AND RESULTS

- if ATM support is not required, then the server can be configured to run without it, as a standard HTTP daemon.

- the size of the server’s footprint may be significantly reduced in memory, through the use of dynamic binding and the operating system’s caching of dynamic libraries and other techniques. This is especially true if there are multiple instances (as opposed to threads) of the server running.

- Bug fixing in a part of the dynamic code can be as simple as replacing the faulty code module and sending the server a reconfigure signal.

The use of the object-oriented binding (inspired by the Abstract Factory design pattern) allows the application to remain completely unaware of the implementation details of the dynamic module—relying only on a published interface.

During the development of the server, the ability to logically separate functionality into network transport, protocol and service modules became apparent. Furthermore, it was possible to dynamically bind to these modules. This allows for a flexibility in configuration of different services with different protocols, all operating over various network transports. The separation of the service module from the protocol allows the independent development of new multimedia content-generation sources, which will dynamically bind with an existing HTTP_ATM server without code change/recompilation.

The use of ATM’s native QoS support was a crucial goal of this design. Through the use of the HTTP PEP protocol, it is possible to negotiate QoS contracts with server supporting QoS, without sacrificing backwards compatibility with traditional HTTP servers and systems.

The concurrency achieved through the use of multi-threading and cached memory-mapping of files ensures that ATM I/O will not suffer as a consequence of inefficient application behaviour. Multi-threading has proven itself in this project to be a powerful development technique, especially when used with kernel-level threads and blocking I/O. The drawbacks of multi-threaded code include the fact that memory mis-management becomes a more serious issue, and access to certain data structures must be properly synchronised to avoid deadlock or conflict.

Through the design and development of the HTTP_ATM server, an appreciation was gained for subtle behavioural interactions between the different components of the system. A macroscopic viewpoint needs to be taken to ensure efficiency, consistency and correctness between network stacks, application level protocols, developer tools and APIs. Optimisations at one level (for example, TCP/IPs complex congestion algorithms) may cause unexpected problems at another level (e.g. for TCP/IP over ATM, or even HTTP/1.0 over TCP/IP).
This chapter introduces the design of a modular, dynamically extensible HTTP server which provides support for ATM services such as QoS negotiation. Depending on its configuration, the server will function as an ordinary (proxying) HTTP server, responding to client requests over TCP/IP.

However, server functionality can be dynamically extended by the addition of extra modules (transports, protocols and services). In addition, the server will interwork requests from one transport through traditional HTTP proxying techniques.

The design introduces the requirement of advanced operating system functionality for the support of these services, notable the requirement of a multi-threading API, and dynamic loading facilities.

In this chapter, the design of the HTTP atm server, as far as possible, is kept separated from the implementation. The impact of implementation techniques on the design is also presented. In particular, the lack of a C++ dynamic loading facility ruled out the possibility of using a class hierarchy of loadable components, as illustrated in Figure 7.3.

In circumventing a lack of functionality in the FORE Systems ATM API, two solutions are presented. The solution which least impacts on the overall design is highlighted, and explained.
Chapter 8

Conclusions

8.1 Overall Conclusions

The purpose of this research was to investigate the provision of access functionality to data services in Broadband ISDN networks. The thesis focused on HTTP, the Hypertext Transfer Protocol use in the World Wide Web as a potential candidate for the coordination of data service access.

A modular design was presented for an application level server offering traditional HTTP processing capabilities, but also HTTP adapted to ATM with Quality of Service support. The design features heavily in the use of threads to achieve server concurrency, and in separation of server functionality into three orthogonal modules - the Service, Protocol and NetTransport - which are dynamically bound with at run-time. Thus the server is dynamically extendible to supporting new network transports and protocols.

As discussed earlier in Section 7.5 (on page 70), modularity in common web servers is restricted to resource name space services. The Apache server has unofficially been modified to allow the dynamic loading of service modules, but again this facility is restricted to resource name space services. The design presented in this thesis outlines a dynamically extensible, modular server with support for multiple protocol revisions and network transports.

In the following sections, ATM will be introduced as a sophisticated networking technology, possibly with technical flaws, and everly lacking in applications and services.

HTTP will be presented as a popular, lightweight resource access protocol. It can easily be extended to support advanced ATM features, such as QoS.

TCP/IP will be presented as the current goal in providing stop-gap ATM applications. It will also be presented as the de facto networking protocol for HTTP.
8.1. OVERALL CONCLUSIONS

The design of HTTP_ATM, a server providing native ATM access to broadband resources will be discussed. Finally, further scope for future study and development will be outlined.

8.1.1 ATM

Asynchronous Transfer Mode was introduced as the networking layer of the Broadband ISDN Reference Model. ATM is a connection-oriented transport technology that utilises a fixed size data unit called a cell to achieve high bandwidth and scalability through hardware switching. The small size of the cell also serves in providing for the low jitter requirements of real time services, and the scalable allocation of network bandwidth to applications. The current cell size has been criticised as being too small to maximise network throughput.

ATM has yet to enjoy large scale deployment. It is a complex technology, whose introduction is being delayed by improvements in the capacity of local area networks, and also in the significant role being played in modern society by computer data networks.

ATM is lacking in application services which show it the potential of its hard quality of service and traffic policing features. As a result, it is imperative that ATM support legacy applications and systems, to ease the migration costs and allow gradual introduction of ATM services.

To make matters worse, the congestion control mechanisms provided by ATM adversely react with similar functionality provided in computer data networks. The slow connection setup times impinge upon the performance of short lived transaction based data transfer.

As a result, it is very likely that the concept of ATM to the end-user desktop is dead. The additional complexity of supporting ATM in the local area network (through the use of emulation and interworking techniques such as Classical IP and LANE) outweighs the cost of over-engineering the network capacity. With the introduction of quality of service features into computer data traffic flows, of isochronous services such as those found in isoEthernet, and the huge capacity of new LAN technologies (such as gigabit ethernet), the role of ATM is being pushed back into the backbone of the network as illustrated in Figure 8.1.

While the provision of HTTP over ATM, with support for QoS, is achievable, ATM is not suitable for short-lived HTTP requests due to call setup delays. ATM does prove useful for QoS enforcement in long-duration calls which transport multimedia data streams. The proxying of HTTP requests from LANs across an ATM WAN environment makes excellent use of the QoS features over the unpredictable WAN environment, whilst allowing the over-engineering of LANs to perform satisfactory “best-effort” delivery to the desktop.
8.1. OVERALL CONCLUSIONS

8.1.2 HTTP

HTTP is developing into a viable application-level control protocol for multimedia streams. With support for pipelined requests, persistent connections, and multiplexing of the resulting (chunked-encoded) request data-streams, HTTP is a promising tool for establishing end-to-end communications in a media rich network.

Section A.2 shows that from locally conducted experiments, the average HTTP response size is somewhere between 5KB and 10KB. Average HTTP request size is difficult to measure, since it is not logged by current HTTP servers.

The concept of achieving separation of stream control and stream data in Broadband networks was introduced. The features of HTTP to support this architecture are examined—its lightweight design, support for arbitrary response types and lengths, its support for content negotiation, its proxying ability to access legacy networks, its pending abilities to provide asynchronous data stream multiplexing and chunked-encoding.

8.1.3 TCP/IP

The Transport Control Protocol of the TCP/IP suite was examined, since it provides the de facto network layer for HTTP-based applications. Its inadequacies in servicing HTTP requirements are discussed—the dynamic congestion control and avoidance algorithms such as slow-startup, the three-way handshake, and the Nagle algorithm.

In addition, the Resource reSerVation Protocol extension to the TCP/IP suite to provide for soft quality of service features was discussed. T/TCP was also introduced as a replacement for TCP in transaction based services. T/TCP cleverly manages to skip the overhead of the initial three-way handshake by means of a special “TAO” test applied to the connection. If the connection fails the
8.2. FUTURE WORK

test, the T/TCP connection gracefully falls over to the normal TCP connection initiation.

8.1.4 HTTP_ATM

The design and implementation of HTTP_ATM was discussed. HTTP_ATM is a dynamically extensible, modular WWW server. It makes use of advanced operating system features such as multi-threading and dynamic loading of object files. It features native support for ATM QoS via simple extensions of HTTP. It also supports legacy clients and systems through the use of HTTP/1.1 proxying.

8.1.5 Design Features and Limitations

The design of HTTP_ATM included some novel features which enhanced its flexibility, and provided compatibility to existing HTTP services. The use of dynamic binding potentially reduces the executable's memory footprint, and provided a significant degree of flexibility in configuration and deployment. Patching the server to fix bugs could be as simple as replacing the component in error, and then sending the server a restart signal.

Supporting ATM's QoS facilities was a priority, whilst maintaining the ability to interwork with legacy HTTP clients and servers. A high-degree of concurrency was achieved through the use of advanced operating system features, such as kernel-level multi-threading, and the use of memory mapping (with a suitable application-level caching strategy).

The subtle interactions of architecture components at various levels were found to often determine the overall performance and correctness of the system.

8.2 Future Work

The performance of the HTTP_ATM server has not fully been tested. A stress test will illustrate various implementation weaknesses, particularly with the simple threading model chosen. In addition, the effect of changing the network transport from TCP/IP to T/TCP has not be examined (T/TCP is not supported in the Solaris environment), especially when running over an ATM link (Classical IP or LANE).

When RSVP matures further, there is potential to investigate the integration of RSVP-based QoS and QoS negotiation into the negotiation facilities already present in HTTP_ATM.

Furthermore, although the design identifies SSCOP as the reliable transfer layer upon AAL5 for HTTP_ATM traffic, the SSCOP has not been implemented. At present, transmission is best-effort over AAL 5, with no error detection or recovery mechanisms in use. This was suitable for the initial
trial and experimentation, but a production server will need reliable transmission capabilities. The SSCOP provides a potential solution to this—however, other solutions should also be identified and considered.

One aspect of traditional web server design which became apparent during the course of this work was the impact that the operating system environment has on server performance.

Figure 8.2 show current arrangement of web server process running upon, for example, a Unix-based operating system. As such, it is subject to process/thread scheduling phenomena, delays in copying data from kernel space to user space, unnecessary duplication of data, lack of integration with network buffers and filesystem cache, and multiple user/kernel boundary crossings with each I/O system call.

There is tremendous potential to investigate the development of a WebOS, promoting the direct integration of basic server functionality into kernel or kernel sub-system. Strategies for implementing this cleanly in both microkernel and macrokernel based environments need to be studied.

The placement of basic web functionality at such a low layer in the operating system makes responding to web requests independent of process scheduling. In addition, it has been empirically demonstrated that up to 80% of latency in servicing web requests is due to I/O - both network and filesystem [HMS97]. The placement of web functionality into kernel space will allow direct access to network buffers (\texttt{sk\_buffers}) and the virtual filesystem cache. This would increase performance in similar manner to increase achieved by kernel implementation of NFS in Linux over the original user-space server. To provide additional functionality such as CGI binaries etc, the use
of portals [Ste95] would allow executables to be associated with requests to certain parts of the filesystem—perhaps even (hopefully) on a kernel level.

The use of dynamic binding provides an interesting opportunity for the management of internet services on Unix. Traditionally, many of these are handled through the use of a special process called the Internet Super-Server \(^1\). This process polls for incoming service request and spawns the appropriate executable to handle the request—after some massaging of the program’s standard in and standard out file descriptors. This is quite expensive in terms of CPU because it is at the heavy-weight process level. There is the potential to use separate threads within the one address space to handle common services, dynamically binding with the appropriate service code at runtime—either when the service is first requested, or through a caching scheme.

The drawbacks to this system include the potential for fragmentation of the process’ heap, and overall system stability would depend on the quality of each service module.

In conclusion, the latencies introduced through the interactions of subsystems and libraries, at different operating system levels, are quite significant in determining overall architecture behaviour. The investigation of tighter integrations of these components, together with unified caching strategies, holds the potential for significantly improved performance. The use of dynamic binding in network server development has proven itself to be a valuable technique, and opportunities for its exploitation should be explored.

\(^1\) Also known as inetd.
Appendix A

HTTP/1.1

A.1 Example HTTP Conversation

It can be difficult to fully understand or follow the workings of HTTP without an example. Table A.1 lists valid HTTP/1.1 header fields, and where they appear or what they can refer to (in the case of response body). Table A.2 lists valid HTTP/1.1 server response codes.

A.2 Average Response Sizes

The average HTTP response size was measured from the log files of three local sites

- the Telecommunications Research Centre web-server (URL: http://www.trc.ul.ie/)

- W3 Services, Ltd. (URL: http://www.commerce.ie/), Internet marketing and website development consultants.

- University of Limerick Computer Society (URL: http://www.csn.ul.ie/)

The sizes are in bytes. While it is strictly incorrect to generalise from analysis of these three sites, they are light to moderately busy web sites. The HTTP response consists of both header information and potentially a MIME-encoded resource.

---

1which mirrors the Virtual Web Museum for Ireland
### Table A.1: HTTP/1.1 Header Fields

<table>
<thead>
<tr>
<th>Name</th>
<th>Client</th>
<th>Server</th>
<th>Body</th>
</tr>
</thead>
<tbody>
<tr>
<td>Accept</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Accept-Charset</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Accept-Encoding</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Accept-Language</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Accept-Ranges</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Age</td>
<td>✓</td>
<td>✓</td>
<td></td>
</tr>
<tr>
<td>Allow</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Authorization</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Cache-Control</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Connection</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-Base</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-Encoding</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-Language</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-Location</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-MD5</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-Range</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Content-Type</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Date</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>ETag</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Expires</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>From</td>
<td>✓</td>
<td>✓</td>
<td>✓</td>
</tr>
<tr>
<td>Host</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>If-Modified-Since</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Last-Modified</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>If-Match</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>If-None-Match</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>If-Range</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Location</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Max-Forwards</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Pragma</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Proxy-Authenticate</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Proxy-Authorization</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Public</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Range</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Referer</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Retry-After</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Server</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Transfer-Encoding</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Upgrade</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>User-Agent</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Vary</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Via</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>Warning</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
<tr>
<td>WWW-Authenticate</td>
<td>✓</td>
<td></td>
<td>✓</td>
</tr>
</tbody>
</table>
A.2. AVERAGE RESPONSE SIZES

<table>
<thead>
<tr>
<th>Code</th>
<th>Description</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>100</td>
<td>Continue</td>
<td>Informational</td>
</tr>
<tr>
<td>101</td>
<td>Switching Protocols</td>
<td></td>
</tr>
<tr>
<td>200</td>
<td>OK</td>
<td>Successful</td>
</tr>
<tr>
<td>201</td>
<td>Created</td>
<td></td>
</tr>
<tr>
<td>202</td>
<td>Accepted</td>
<td></td>
</tr>
<tr>
<td>203</td>
<td>Non-Authorative Information</td>
<td></td>
</tr>
<tr>
<td>204</td>
<td>No Content</td>
<td></td>
</tr>
<tr>
<td>205</td>
<td>Reset Content</td>
<td></td>
</tr>
<tr>
<td>206</td>
<td>Partial Content</td>
<td></td>
</tr>
<tr>
<td>300</td>
<td>Multiple Choices</td>
<td>Redirection</td>
</tr>
<tr>
<td>301</td>
<td>Moved Permanently</td>
<td></td>
</tr>
<tr>
<td>302</td>
<td>Moved Temporarily</td>
<td></td>
</tr>
<tr>
<td>303</td>
<td>See Other</td>
<td></td>
</tr>
<tr>
<td>304</td>
<td>Not Modified</td>
<td></td>
</tr>
<tr>
<td>305</td>
<td>Use Proxy</td>
<td></td>
</tr>
<tr>
<td>400</td>
<td>Bad Request</td>
<td>Client Error</td>
</tr>
<tr>
<td>401</td>
<td>Unauthorized</td>
<td></td>
</tr>
<tr>
<td>402</td>
<td>Payment Required (reserved)</td>
<td></td>
</tr>
<tr>
<td>403</td>
<td>Forbidden</td>
<td></td>
</tr>
<tr>
<td>404</td>
<td>Not Found</td>
<td></td>
</tr>
<tr>
<td>405</td>
<td>Method Not Allowed</td>
<td></td>
</tr>
<tr>
<td>406</td>
<td>Not Acceptable</td>
<td></td>
</tr>
<tr>
<td>407</td>
<td>Proxy Authentication Required</td>
<td></td>
</tr>
<tr>
<td>408</td>
<td>Request Timeout</td>
<td></td>
</tr>
<tr>
<td>409</td>
<td>Conflict</td>
<td></td>
</tr>
<tr>
<td>410</td>
<td>Gone</td>
<td></td>
</tr>
<tr>
<td>411</td>
<td>Length Required</td>
<td></td>
</tr>
<tr>
<td>412</td>
<td>Precondition Failed</td>
<td></td>
</tr>
<tr>
<td>413</td>
<td>Request Entity Too Large</td>
<td></td>
</tr>
<tr>
<td>414</td>
<td>Request-URI Too Long</td>
<td></td>
</tr>
<tr>
<td>415</td>
<td>Unsupported Media Type</td>
<td></td>
</tr>
<tr>
<td>500</td>
<td>Internal Server Error</td>
<td>Server Error</td>
</tr>
<tr>
<td>501</td>
<td>Not Implemented</td>
<td></td>
</tr>
<tr>
<td>502</td>
<td>Bad Gateway</td>
<td></td>
</tr>
<tr>
<td>503</td>
<td>Service Unavailable</td>
<td></td>
</tr>
<tr>
<td>504</td>
<td>Gateway Timeout</td>
<td></td>
</tr>
<tr>
<td>505</td>
<td>HTTP Version Not Supported</td>
<td></td>
</tr>
</tbody>
</table>

Table A.2: HTTP/1.1 Status Codes
### A.2. AVERAGE RESPONSE SIZES

<table>
<thead>
<tr>
<th>Site</th>
<th>Server Software</th>
<th>Requests</th>
<th>Total Size</th>
<th>Average Size</th>
</tr>
</thead>
<tbody>
<tr>
<td>W3 Services</td>
<td>Netscape Enterprise/2.0d</td>
<td>202315</td>
<td>988212598</td>
<td>4884.52</td>
</tr>
<tr>
<td>TRC</td>
<td>Netscape-Communications/1.1</td>
<td>413716</td>
<td>4317163357</td>
<td>10435.09</td>
</tr>
<tr>
<td>Comp. Soc.</td>
<td>Apache/1.2b10</td>
<td>389442</td>
<td>3339365568</td>
<td>8574.74</td>
</tr>
<tr>
<td><strong>Total:</strong></td>
<td></td>
<td>1005473</td>
<td>8644741523</td>
<td>8597.69</td>
</tr>
</tbody>
</table>

#### A.2.1 Analysis Script

The following Perl script was used to perform the analysis on the web server log files, which logged in the common log file format.

```perl
#!/usr/local/bin/perl

# SCRIPT: average.pl
# AUTHOR: Ivan Griffin (ivan.griffin@ul.ie)
# DATE: Mon Jan 20 17:38:48 GMT 1997
#
# REVISION HISTORY:
# 10 Jul 1996 IJG Original version (access.pl)
# 20 Jan 1997 IJG Modified from access.pl source
#
#
# outlaw barewords and set up the paranoid stuff
#
use strict 'subs';
use English;

$ENV{'PATH'} = '/bin:/usr/bin:/usr/ucb';
$ENV{'IFS'} = '';

# some initial values and script defines

$TotalHits = 0;
$TotalSize = 0;
$AverageSize = 0;

#
# process logfiles (stdin)
#
sub process_data
{
```
while (<STDIN>)
{
    chop;
    ($_, $_, $_, $_, $_, $_, $_, $Status, $Size) = split(' ',$_);
    if ($Status == 200)
    {
        $TotalSize += $Size;
        $TotalHits ++;
    }
};

$AverageSize = $TotalSize / $TotalHits;

print <<EOF;
The total number of requests is $TotalHits.
The total size of all bytes transferred is $TotalSize.
The average file size is $AverageSize.
EOF
}

# main()
#

&process_data;
Appendix B

OMT

B.1 Introduction

This section describes the use of the OMT object model notation in the design chapter of this thesis (Chapter 7).

B.2 Diagrammatic Notation

Figure B.1 shows the diagrammatic notation used in the object models in this thesis. Some criticisms of the notation are also presented, with potential solutions requiring notational changes.

The OMT object model deals with the static relationships between *classes of objects*. The term class is used in the categorisation of system objects—each object is an *instance* of a particular class.

Classes are described by a rectangle divided into three sections - showing the class name, class operations and class fields. An abstract class is one in which some or all of the operations have an `{abstract}` attribute.

Lines between classes depict class associations, showing how the framework is designed to be used. This is an extremely powerful design notation to illustrate framework invariants—it is not enough to provide a framework of class interfaces without also specifying how the classes within the framework are to interact.

Associations may be qualified—for example, Figure B.2 (this figure is taken from Figure 7.12, on page 89). In this example, the ConnectionMonitor class monitors the existence of the AtmTransport class using a timer. When the timer expires, the ConnectionMonitor class will kill the thread executing in the monitored AtmTransport instance.

The triangle symbol illustrates class inheritance—a *subclass* is said to “inherit” certain features
Figure B.1: OMT Object Model Notation
B.3. OBJECT MODEL WEAKNESSES

of its super-class. It can also override other features and functionality. The subclass is therefore a more specialised form of the super-class. Inheritance is also known as the “IS-A” relationship. Any class wishing to inherit from an abstract class must provide an implementation for all the abstract operations.

The diamond symbol illustrates class aggregation. When one class aggregates another, the subordinate class describes some part of the superior class. A class may be composed of a number of subordinate classes. For example, an automobile class is not a type of colour (not an “IS-A” relationship), yet an automobile can be described as having a certain colour. Thus, the aggregation relationship is also known as the “HAS-A” relationship.

Finally, a dotted line leading from a class operation to a rectangle with a “folded” top right-hand corner indicates an implementation of the class method.

B.3 Object Model Weaknesses

Traditional OMT [RBP+91] does not have directed associations. It also places class attributes above class methods in class diagrams. Personally, I disagree with this, and consider the contract a class makes through its interface (its methods) to be more important than internal instance fields or variables. Likewise, directed associations are sometimes quite useful in clearly illustrating role relationships in an association. The modified OMT object-model notation used in this thesis is also used in [GHJV94].

Often, there are times when, to encapsulate a complete behaviour or interface, methods and fields which are not instance-specific but operate on the class as a whole are necessary. In languages where the classes themselves are objects, implementing this is quite clean. In C++, it required adding additional semantics to the use of the static keyword. The OMT object model does not
B.3. OBJECT MODEL WEAKNESSES

provide any means of clearly identifying the class wide elements in the interface. Perhaps the use of a \{static\} attribute on the class member could server this purpose, similar to the way the \{abstract\} attribute is currently used.

The OMT object model does not support differentiation between private (implementation specific) class members, and public class members (part of the class interface). This is a serious flaw in the OMT object model notation.

A potential solution to this is to create a separation of the object model into two layers of detail:

- a primary level showing the class name, static methods and fields, class public methods and class associations. Publicly accessible class fields may also be shown here in same section as methods. Methods are identified by use of parenthesis after the name (\texttt{operation()}).

- a secondary level, building on level 1 by showing private methods. I do not believe that there is a need to show private class instance variables, as they do not form part of the class interface, and are strictly an implementation issue.

As an example of this, the level structures are shown in Figure B.3. With the use of transparencies et cetera, it would be possible to overlay the secondary level detail upon the primary level diagram. I believe this would significantly improve the quality of the information in the object model. At its most fundamental level, it would convey the interfaces the various classes in the system use to interact and nothing more.
Appendix C

Sample Source Code

C.1 Introduction

This appendix describes the various application program interfaces used in the development of HTTP_ATM, explaining the salient features and usage in the context of the architecture presented in Figure 7.9.

Section C.1 details the platform upon which the server was developed and tested, and the tools used in the development process. It was originally envisaged that the experimental server would be built upon the ATM-testbed presented in Figure C.1. However, due to the unavailability of firmware with support for SVCs for the Tellabs Alta 2600 switch, it was necessary to develop using a point-to-point fibre connection directly between the Sun SPARCstations.

<table>
<thead>
<tr>
<th>Operating System</th>
<th>Solaris 2.5.1 / CDE 1.0.1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Platform</td>
<td>Sun SPARCstation 5</td>
</tr>
<tr>
<td>ATM NIC</td>
<td>FORE Systems SBA-200</td>
</tr>
<tr>
<td>Signalling Protocol</td>
<td>FORE SPANS</td>
</tr>
<tr>
<td>Firmware Release</td>
<td>A_ForeThought_4.0.0 (1.37)</td>
</tr>
<tr>
<td>Development Tools</td>
<td>GNU gcc version 2.7.2, libg++ 2.7.2.1, SunPRO SC4.0 18 Oct 1995 C++ 4.1 GNU Make version 3.71 GNU gdb 4.16 ddd 2.0 PureAtria Purify 4.0.1</td>
</tr>
</tbody>
</table>

Table C.1: Development Environment
C.2 Solaris Thread API

When dealing with multiple threads of execution within a single process, there are two possibilities for positioning thread scheduling:

• in user-space: In this case, the threads are multiplexed within the one process. Switching from one to the next does not require a user-kernel space crossing and thus this provides the fastest possible threading implementation. However, this threading model causes all threads to block when any thread makes a blocking I/O system call. As such, it is generally reserved for primarily CPU-intensive applications.

• in kernel-space. Since the thread scheduling is done in kernel space threads which make blocking I/O calls will not affect others in the same process space. However, scheduling of threads now requires a user-kernel space crossing (generally, a switch from a protected mode in the processor to supervisor mode), which has a larger overhead than user-space scheduling.

Most thread APIs provide either a user-space or kernel-space implementation of threading.

The Solaris multi-threaded architecture is one of the most functional thread architectures supported by any operating system. Both user-space and kernel-space scheduling of threads is supported. Light-weight user-level threads are multiplexed upon kernel supported threads of control. This allows the program developer the greatest possible flexibility in separating logical (program)
concurrency from real concurrency (via kernel-threads, and thus relatively costly) and control both within a single programming model[PKB+91].

Figure C.2 shows the two levels of the thread model. A process in Solaris is a virtual memory space, and an associated set of (kernel-level) light-weight processes (LWP), which are individually schedulable contexts-of-execution. Each user-level thread runs in a LWP. All the threads and LWPs accessing the same address space reside in a single process—LWPs are not shared between processes.

Depending on the hardware configuration, the Solaris scheduler will try to maximise concurrency by assigning an LWP to each idle CPU.

C.2.1 Sample Code

The following code excerpt shows the creation of a thread to do some work, how to protect access around shared (global) data, and how to join (synchronise) with the thread when it has finished its work.

```c
#define _REENTRANT
#include <assert.h>

extern "C"
{
    #include <thread.h>
}

bool workDone = false;
```
mutext_t myMutex = { 0 };

void* _WorkFunction(void*);

// some function
{
    int status =
        mutex_init(&myMutex, USYNC_THREAD, NULL); // intra-process mutex

    thread_t myThreadId = 0;
    status = thr_create(0, // stack base
                        0, // thread stack size
                        _WorkFunction, // function to start executing
                        0, // function argument
                        0, // flags
                        &myThreadId); // thread identifier

    assert(0 == status);

    // wait for thread to complete

    status = thr_join(myThreadId, 0, 0);
    assert (0 == status);

    // rest of function

    
}

void* _WorkFunction(void*)
{
    // protect access to workDone variable

    int status = mutex_lock(&myMutex);
    assert(0 == status);

    if (true == workDone) thr_exit((void*) 0);

    status = mutex_unlock(&myMutex);
    assert(0 == status);

    // do some work

    for (int i = 0; i <10000; i++)
        ;

    status = mutex_lock(&myMutex);
    assert(0 == status);
C.3. SIGNAL HANDLING CODE FOR THREADS

workDone = true;
status = mutex_unlock(&myMutex);
assert(0 == status);

return ((void*) 0);
}

C.3 Signal Handling Code for Threads

The listing below shows some skeleton C++ code for implementing the creation of a signal handler thread (as used in the httpDaemon class) using Solaris' native threads.

C.3.1 Source to establish a signal handler thread

//
// AUTHOR: Ivan Griffin (ivan.griffin@ul.ie)
// DATE: Monday, April 7, 1997
// PURPOSE: C++ Code fragment illustrating creation of a signal handler thread using Solaris native-thread API.
//
#define _REENTRANT

extern "C"
{
#include <thread.h>
#include <signal.h>
#include <sys/types.h>

void* _MySignalHandler(void*);

int main(void)
{
    sigset_t set = { { 0 } };

    // block all signals in current thread -
    // creates created by this thread will inherit the
    // same signal mask
    thr_sigsetmask(SIG_SETMASK, &set, (sigset_t*) 0);

    // create a special purpose signal handler thread, to catch
    // all signals and decide what to do with them.
    // Certain asynchronous signals (such as SIGFPE from division


The most popular method of TCP/IP programming is to use the “BSD Socket” interface [LFJ+], developed by the University of California at Berkeley Computer Systems Research Group (UCB-CSRG). Within the programming abstraction presented by this C language API, network endpoints (the tuples composed of IP address and port number) are represented as sockets.

The socket inter-process communication facilities (introduced with the release of the 4.2BSD operating system) were designed to allow network-based applications to be constructed independently of the underlying communication facilities.

The BSD socket interface has become a de facto standard in the industry, and greatly helped establish TCP/IP as a world leading network protocol. HTTP is most commonly implemented over
C.4. BSD SOCKETS

TCP/IP, by availing of the BSD Socket API [FGM+97].

C.4.1 Creating a Server

To create a server application using the BSD interface, the following steps are necessary:

1. create a new socket (using the socket() system call).

2. bind an address (IP address and port number) to the socket. This step identifies the server so that the client knows where to go (using the bind() system call).

3. listen for new connection requests on the socket (using the listen() system call).

4. accept new connections (using the accept() system call).

Addresses are expressed in “network format”, and thus may require conversion from the native byte-order of the host operating system. C macro functions are provided for this task.

Often, the servicing of a request on behalf of a client may take some considerable amount of time. It would be more efficient in such a case to be able to accept and deal with new connections while a request is being processed. The most common way of doing this is for the server to fork a new copy of itself after accepting the new connection. However, this introduces the extra overhead of new process creation, and context switching. More recently, servers tend to use multiple threads instead of performing a fork. This was the approach taken in the design of the HTTP_ATM server.

Using multiple threads instead of multiple processes may lighten the load on the server host, thereby increasing efficiency. Context-switching between threads (in the same process address space) generally has much less associated overhead than switching between different processes. However, since most of the slave threads in this case are doing network I/O, they must be kernel level threads. If they were user-level threads, the first thread to block on I/O would cause the whole process to block. This would result in starving all other threads of any CPU attention until the I/O had completed.

C.4.2 Server Sample Code

/*
 * MODULE: sampleServer.c
 * AUTHOR: Ivan Griffin (ivan.griffin@ul.ie)
 * DATE: Thursday, February 6, 1997
 *
 * PURPOSE: Example C code to demonstrate creation of a
 * multi-threaded TCP/IP server using the BSD Socket API
 */
#include <stdio.h> /* */
#include <stdlib.h> /* exit() */
#include <string.h> /* memset(), memcpy() */
#include <sys/utsname.h> /* uname() */
#include <sys/types.h>
#include <sys/socket.h> /* socket(), bind(), listen(), accept() */
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>
#include <unistd.h> /* fork(), write(), close() */
#include <thread.h> /* thr_create(), thr_exit() */

/ *
* prototypes
*/
int _GetHostName(char *buffer, int length);

/ *
* constants
*/
const char MESSAGE[] = "Hello, World!\n";
const int BACK_LOG = 5;

int main(int argc, char *argv[])
{
  int serverSocket = 0,
      on = 0,
      port = 0,
      status = 0,
      childPid = 0;
  struct hostent *hostPtr = NULL;
  char hostname[80] = "";
  struct sockaddr_in serverName = { 0 };

  if (2 != argc)
  {
    fprintf(stderr, "Usage: %s <port>\n", argv[0]);
    exit(1);
  }
  port = atoi(argv[1]);

  serverSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
  if (-1 == serverSocket)
  {
    perror("socket()");
    exit(1);
  }

  /*
  * turn off bind address checking, and allow port numbers
to be reused — otherwise the TIME_WAIT phenomenon will
prevent binding to these address.port combinations for
(2 * MSL) seconds.
*/

on = 1;

status = setsockopt(serverSocket, SOL_SOCKET, SO_REUSEADDR,
(const char *) &on, sizeof(on));

if (-1 == status)
{
    perror("setsockopt(...,SO_REUSEADDR,...)");
}

/*
 * when connection is closed, there is a need to linger to ensure
 * all data is transmitted, so turn this on also
 */
{
    struct linger linger = { 0 };
    linger.l_onoff = 1;
    linger.l_linger = 30;
    status = setsockopt(serverSocket, SOL_SOCKET, SO_LINGER,
    (const char *) &linger, sizeof(linger));

    if (-1 == status)
    {
        perror("setsockopt(...,SO_LINGER,...)");
    }
}

/*
 * find out who I am
 */
status = _GetHostName(hostname, sizeof(hostname));
if (-1 == status)
{
    perror("_GetHostName()");
    exit(1);
}

hostPtr = gethostbyname(hostname);
if (NULL == hostPtr)
{
    perror("gethostbyname()");
    exit(1);
}

(void) memset(&serverName, 0, sizeof(serverName));
(void) memcpy(&serverName.sin_addr, hostPtr->h_addr,
hostPtr->h_length);
/*
 * to allow server be contactable on any of its
 * IP addresses, uncomment the following line of code:
 *
 * serverName.sin_addr.s_addr = htonl(INADDR_ANY);
 */

serverName.sin_family = AF_INET;
serverName.sin_port = htons(port); /* network-order */

status = bind(serverSocket, (struct sockaddr *) &serverName,
              sizeof(serverName));
if (-1 == status)
    perror("bind()");
    exit(1);

status = listen(serverSocket, BACK_LOG);
if (-1 == status)
    perror("listen()");
    exit(1);

for (;;)
{
    struct sockaddr_in clientName = { 0 };
    int slaveSocket, clientLength = sizeof(clientName);
    (void) memset(&clientName, 0, sizeof(clientName));

    slaveSocket = accept(serverSocket,
                         (struct sockaddr *) &clientName, &clientLength);
    if (-1 == slaveSocket)
    {
        perror("accept()");
        exit(1);
    }

    thr_create((void*) 0, 0, _HandleRequest, &slaveSocket, THR_DETACHED,
                (void*) 0);
}

return 0;

void* _HandleRequest(void *arg)
{
    int slaveSocket = *((int*) arg);

    if (-1 == getpeername(slaveSocket,
                (struct sockaddr *) &clientName, &clientLength))
    {

```c
perror("getpeername()"既然
} else
{
    printf("Connection request from %s\n",
           inet_ntoa(clientName.sin_addr));
}

/*
 * Server application specific code goes here,
 * e.g. perform some action, respond to client etc.
 */
write(slaveSocket, MESSAGE, strlen(MESSAGE));
close(slaveSocket);
thr_exit(0);
/* Never Reached */
return ((void*) 0);
}

/*
 * Local replacement of gethostname() to aid portability
 */
int _GetHostName(char *buffer, int length)
{
    struct utsname sysname = { 0);
    int status = 0;

    status = uname(&sysname);
    if (-1 != status)
    {
        strncpy(buffer, sysname.nodename, length);
    }

    return (status);
}

C.4.3 Creating a Client

In the following client example, a socket is created like before. The first command line argument is first of all assumed to be a hostname for the purposes of finding the server's address. If this fails, it is assumed to be a dotted quad IP address. If this also fails, then the client cannot resolve the server's address and will not be able to contact it.

Having located the server, an address structure is created for the client socket. No explicit call to bind() is needed here, the connect() call handles all this.

Once the connect() returns, all going well, a duplex connection will now have been established.
```
Like the server, the client can now use `read()` and `write()` calls to receive/send data on the connection. There are a number of issues to be aware of when sending binary data over a socket, in particular relating to byte-ordering, character set conversions (ASCII / EBDIC / Unicode / et cetera), attempting to pass memory pointers and file descriptors between processes etc.

Additionally, you must ensure that you handle short `write()`'s correctly. Short writes happen when the `write()` call only partially writes a buffer to a file descriptor. They occur due to buffering in the operating system, and due to flow control in the underlying transport protocol. Certain system calls, termed slow system calls, may be interrupted. Some may or may not be automatically restarted, so you should explicitly handle this when network programming.

It is common practice to close unnecessary socket file descriptors in child and parent processes (in both servers and clients) when using the simple forking model. This prevents the child or parent from potential erroneous reads or writes (due to coding error), and also frees up descriptors, which are a limited resource. Do not try this when using threads! Multiple threads within a process share the same memory space and set of file descriptors. If the server socket is closed in a slave thread, then it is closed for all other threads in that process!

### C.4.4 Client Sample Code

```c
#include <stdio.h> /* perror() */
#include <stdlib.h> /* atoi() */
#include <sys/types.h>
#include <sys/socket.h>
#include <unistd.h> /* read() */
#include <netinet/in.h>
#include <arpa/inet.h>
#include <netdb.h>

int main(int argc, char *argv[
{ 
    int clientSocket,
    remotePort,
    status = 0;
    struct hostent *hostPtr = NULL;
    struct sockaddr_in serverName = { 0 }; 
    char buffer[256] = "";
```
char *remoteHost = NULL;
if (3 != argc)
{
    fprintf(stderr, "Usage: %s <serverHost> <serverPort>\n", argv[0]);
    exit(1);
}
remoteHost = argv[1];
remotePort = atoi(argv[2]);

clientSocket = socket(PF_INET, SOCK_STREAM, IPPROTO_TCP);
if (-1 == clientSocket)
{
    perror("socket()");
    exit(1);
}

/*
 * need to resolve the remote server name or IP address
 */
hostPtr = gethostbyname(remoteHost);
if (NULL == hostPtr)
{
    hostPtr = gethostbyaddr(remoteHost, strlen(remoteHost), AF_INET);
    if (NULL == hostPtr)
    {
        perror("Error resolving server address ");
        exit(1);
    }
}

serverName.sin_family = AF_INET;
serverName.sin_port = htons(remotePort);
(void) memcpy(&serverName.sin_addr, hostPtr->h_addr, hostPtr->h_length);

status = connect(clientSocket, (struct sockaddr*) &serverName, sizeof(serverName));
if (-1 == status)
{
    perror("connect()");
    exit(1);
}

/*
 * Client application specific code goes here
 *
 * e.g. receive messages from server, respond, etc.
 */
while (0 < (status = read(clientSocket, buffer, sizeof(buffer) - 1)))
printf("%d: %s", status, buffer);
}

if (-1 == status)
{
    perror("read()");
}

/*
 * Will not attempt to write message back to the server
 * Handling short writes
 */
{
    int bytesToSend = status,
        bytesWritten = 0,
        num = 0;

    /*
     * somewhere here bytesToSend, buffer, and fileDesc must be
     * set up.
     */
    for (bytesWritten = 0; bytesWritten < bytesToSend;
         bytesWritten += num)
    {
        num = write(clientSocket,
                    (void *)( (char *)buffer + (char *)bytesWritten ),
                    bytesToSend - bytesWritten);

        if (num < 0)
        {
            perror("write()");

            if (errno != EINTR)
            {
                exit(1);
            }
        }
    }
}

close(clientSocket);

return 0;
C.5 Dynamic Linking and Loading

C.5.1 Introduction to Dynamic Linking

The Executable and Linking Format (ELF) is a binary object file format, originally developed by UNIX Systems Laboratories (USL) for SVR4 UNIX. It is the default binary format for the executables of SVR4 (from various vendors), Sun Solaris and Linux, amongst others.

With ELF, there are two distinct methods of linking with libraries - static linking and dynamic linking. Static linking is the simplest. The libraries that a program uses are combined with the code of the application itself - the result is a self-contained executable.

However, statically linking the same libraries again and again with each separate executable is wasteful - both of disk space and also of memory (a precious resource). Hence, the introduction of dynamic linking facilities.

With dynamic linking, the compiled application binds with the shared library code at run-time. The resulting application is much smaller in terms of the number of sectors it uses on disk, and also in its memory footprint. There is now only one copy of the library code on disk for all applications that use it (by dynamically linking), instead of it being combined with each application and stored multiple times (as is the case with static linking).

With memory usage, the situation is a little different. Solaris manages memory as finely grained allocations called pages. When an application binds at run-time to a library, the library’s code and data segments are loaded into memory if they do not already reside there. The linking is then performed using COPY_ON_WRITE semantics.

The virtual memory system gives each application its own private address space. With COPY_ON_WRITE, the library segments are initially mapped into the application’s address space in a read-only shared fashion. If, at some later stage, the application tries to write to part of these shared memory segments (for example, to change or set the value of a library variable), a special exception will be raised for the offending memory address (called a page fault). The offending page will be copied and privately mapped into the application’s address space, before the offending instruction is re-tried.

In this manner, not only will multiple applications share a single a single copy of a library on disk, but it is very likely that a high proportion of the library will be shared at run-time, thereby reducing the overall memory requirements for the system.

Along with the considerable advantages of an economical, prudent use of system resources, dynamic linking brings with it a few additional complexities compared with static linking. When
an application is run, it may not necessarily find all the libraries it requires. If so, it will die with an error message indicating that it could not find the required library. Similarly, an application may find different versions of the libraries to the versions it was build with.

Libraries, by convention are named with both major and minor revision numbers. An increment in a minor number indicates small changes in the library - perhaps a bug fix or two, or relatively minor new features. Programs are not expected to break as a result of minor number changes.

Major number increments signify large structural or implementation changes to a library - to such a degree that programs linked with a lower major version of the library would not be expected to work with the newer version.

C.5.2 Dynamic Loading

With ELF, the dynamic linking tools allow arbitrary ELF binary objects to be accessed\(^1\). The procedure is quite simple:

1. open a "handle" to an ELF shared library. (If the library contains an _init function, it is invoked when the archive is first opened.)
2. look-up and bind to a particular symbol (variable / function)
3. use the symbol in your code
4. close the handle, unloading the library if necessary. (If the archive contains a _fini routine, it is invoked just before the archive is unloaded.)

The sample code below shows the dynamic invocation of a function in an ELF shared library. First, a handle to the object is obtained using the `dlopen()` system call. The first argument is the filename of the archive. If this is not an absolute pathname, the search for the library will check a colon-separated list of directories in the environment variable `LD_LIBRARY_PATH`. If other shared objects were linked with the object when the object was built (i.e., the object has dependencies on other shared objects), those objects will automatically be loaded by `dlopen()`.

If the filename is `NULL`, a handle is returned for the current program. The handle should not be interpreted or manipulated directly by the application programmer. By convention, dynamically linked libraries and shared objects are named with the prefix `lib` and the suffix `.so`.

After the filename, the next parameter is a flag specifying when the symbols from the library are to be resolved - either `RTLD_LAZY` (when the object is first used) or `RTLD_NOW` (immediately) are valid.

\(^1\) Or at least Solaris 2.x does (the dynamic link features owe their origins to Solaris). Linux however only seems to like providing this feature for ELF shared libraries.

126
If \texttt{RTLD\_GLOBAL} is bitwise-OR'ed with flag, external symbols defined in the library will be made available to subsequently loaded libraries. If the \texttt{testHarness} program is compiled by passing the \texttt{-rdynamic} command line option to the linker (e.g. \texttt{gcc -rdynamic ...} or \texttt{ld -rdynamic ...}), global symbols in the testcode executable will also be used to resolve symbols in the \texttt{dlopen()}'ed object.

Error messages are obtainable in the form of human-readable strings via the \texttt{dlerror()} call.

The \texttt{dlsym()} function takes a valid handle and a \texttt{NULL} terminated symbol name. It returns the address where the symbol is loaded. If the symbol could not be found, \texttt{dlsym()} will return \texttt{NULL}. However, the value of the symbol may actually be \texttt{NULL}. \texttt{dlerror()} returns a valid address if an error did occur, otherwise it returns \texttt{NULL}. Therefore, to ensure an error has occurred, the value returned by \texttt{dlerror()} must be tested to ensure it is not \texttt{NULL}.

It is necessary to store the value returned by \texttt{dlerror()} into a variable, as subsequent calls to \texttt{dlerror()} will return \texttt{NULL} (and if it had not been stored, the error message would have been lost).

Finally, \texttt{dlclose()} is called when the object’s symbols are no longer required.

C.5.3 Sample Test Harness C Source

```c
/*
 * MODULE: testHarness.c
 * AUTHOR: Ivan Griffin (ivan.griffin@ul.ie)
 * DATE: Friday, February 21, 1997
 * PURPOSE: Example C code to demonstrate dynamic loading
 * using ELF dlsym API - test harness
 */

#include <stdio.h>
#include <stdlib.h>
#include <dlfcn.h> /* for dynamic loading functions */

int main(void)
{
    void* handle = 0;
    void (*helloWorld)(void) = 0;
    const char* error = 0;

    handle = dlopen("./libHelloWorld.so", RTLD\_LAZY);
    if (0 == handle)
    {
        error = dlerror();
        if (0 != error)
        {
            fprintf(stderr, "%s: %s\n", argv[0], error);
        }
    }
    return 0;
}
```

127
C.6. FORE ATM API

The FORE Systems ATM API is a user-level API which allows the development of applications on an ATM network. The library uses a STREAMS interface to communicate with the ATM device driver. Communication is via a connection-oriented client-server model. The data transfer phase of
the model is best effort delivery.

ATM applications are addressed in the API by a *Network Service Access Point* (NSAP) which uniquely identifies the end-system, and an *Application Service Access Point* (ASAP).

The library uses the *Simple Protocol for ATM Network Signaling* (SPANS) signalling protocol for connection establishment. SPANS is a protocol developed by FORE Systems, Inc.

Applications first open a file descriptor using the `atm_open()` library call, and then associate a local ASAP with it using `atm_bind()`. At this time, the local NSAP is implicitly bound with the same file descriptor.

Clients initiate an active connection using the `atm_connect()` function, whereas servers use `atm_accept()`. Dataflow may be simplex or duplex. The API supports the specification of call QoS parameters by the agent initiating the active connection (i.e. the client).

Incredibly, the delete keyword from C++ is used in the header files of the API. This name clash must be circumvented by means of the preprocessor before the API can be used in C++ code.

C.6.1 Client Sample Code

This section includes sample C++ code for a client application which uses AAL 5 as the adaptation layer.

```c++

// // MODULE: atmClient.cc
// // AUTHOR: Ivan Griffin (ivan.griffin@ul.ie)
// // DATE: Friday, February 21, 1997
// // PURPOSE: Demonstrate the use of the FORE Systems ATM API
// //

#include <iostream.h>
#include <strstream.h>
#include <assert.h>

extern "C"
{
    #include <stdio.h>
    #include <stdlib.h>
    #include <sys/types.h>
    #include <unistd.h>
    #include <sys/file.h>
    #include <sys/fcntl.h>
    #include <sys/types.h>

    #define delete _delete
```
```c
#include <fore/types.h>
#include <fore_atm/fore_atm_user.h>
#undef delete
}

int main(int argc, char* argv[]) {
    int initialFd, clientFd, i, j, mtu, queueLen;
    Atm_info info;
    Atm_endpoint dst;
    Atm_qos qos;
    Atm_qos_sel qos_selected;
    Atm_sap serviceSap;
    Aal_type aal = aal_type_5;
    Atm_dataflow dataflow = duplex;
    char* rbuf = 0;
    const int SERVER_SAP = 8080;
    
    const char* device = "/dev/fa0";
    
    if (3 > argc) {
        cerr << "Usage: " << argv[0] << " device hostname [server-sap]" << endl;
        exit(1);
    }
    initialFd = atm_open(device, O_RDWR, &info);
    assert(0 <= initialFd);
    mtu = info.mtu;
    rbuf = new char[mtu]; assert(0 != rbuf);
    serviceSap = 0;
    int status = atm_bind(initialFd, serviceSap, &serviceSap, 0);
    if (0 > status) {
        // atm_errro("atm_bind");
        cerr << "atm_bind()" << endl;
        exit(1);
    }
    
    cout << "SAP Assigned is " << serviceSap << endl;
    status = atm_gethostbyname(argv[2], &dst.nsap);
    if (0 > status) {
        cerr << "atm_gethostbyname() failed" << endl;
        exit(1);
    }
    
    if (4 == argc) {
        dst.asap = atoi(argv[3]);
    }
}
```
else
{
    dst.asap = SERVER_SAP;
}

qos.peak_bandwidth.target = 0;
qos.peak_bandwidth.minimum = 0;
qos.mean_bandwidth.target = 128;
qos.mean_bandwidth.minimum = 64;
qos.mean_burst.target = 2;
qos.mean_burst.minimum = 1;

status = atm_connect(initialFd, &dst, &qos, &qos_selected, aal, dataflow);
if (0 > status)
{
    atm_error("atm_connect()");
    exit(1);
}

status = atm_recv(initialFd, rbuf, mtu-1);
if (0 > status)
{
    atm_error("atm_recv()");
    exit(1);
}

cout << rbuf << endl;
atm_close(initialFd);

u_int newSap = atoi(rbuf);

int newFd = atm_open(device, O_RDWR, &info);
assert(0 <= newFd);

status = atm_bind(newFd, newSap, &newSap, 0);
if (0 > status)
{
    atm_error("atm_bind()");
    exit(1);
}

status = atm_connect(newFd, &dst, &qos, &qos_selected, aal, dataflow);
if (0 > status)
{
    atm_error("atm_connect()");
    exit(1);
}

atm_send(newFd, "Hi", 2);
atm_close(newFd);
C.6.2 Server Code

This section includes sample C++ code for the application server, again using AAL 5 as the adaptation layer.

```c++
#include <iostream.h>
#include <strstream.h>
#include <assert.h>

extern "C"
{

#include <stdio.h>
#include <stdlib.h>
#include <sys/types.h>
#include <unistd.h>
#include <string.h>
#include <sys/file.h>
#include <sys/fcntl.h>

#define delete _delete
#include <fore/types.h>
#include <fore_atm/fore_atm_user.h>
#undef delete

int main(int argc, char* argv[])
{
    int mainFd, slaveFd, mtu, queueLen = 20, connectionId;
    u_int switchId, portId;

    Atm_info info;
    Atm_endpoint calling;
    Atm_qos qos;
    Atm_sap serverSap;
    Aal_type aal;
    Atm_dataflow dataflow = duplex;
    const char* deviceName = "/dev/fa0";
    const int SERVER_SAP = 8080;
    int status;
    return 0;
}
```
char* tbuf, * rbuf;

if (2 > argc)
{
    cerr << "Usage: " << argv[0] << " device [server-sap]" << endl;
    exit(1);
}

mainFd = atm_open(deviceName, O_RDWR, &info);
assert (0 <= mainFd);

mtu = info.mtu;
// tbuf = new char[mtu]; assert(0 != tbuf);
rbuf = new char[mtu]; assert(0 != rbuf);

memset(rbuf, 0, mtu);

if (3 == argc)
{
    serverSap = atoi(argv[2]);
}
else
{
    serverSap = SERVER_SAP;
}

status = atm_bind(mainFd, serverSap, &serverSap, queueLen);
if (0 > status)
{
    atm_error("atm_bind()");
    exit(1);
}

cerr << "Server SAP assigned=": " << serverSap << endl;

status = atm_listen(mainFd, &connectionId, &calling, &qos, &aal);
assert(0 <= status);

GET_SWITCH(switchId, calling.nsap);
GET_PORT(portId, calling.nsap);

cerr << "Calling Switch:" " << endl
    << "\tSwitch = " << switchId << " Port = " << portId
    << " SAP = " << calling.asap << " aal = " << aal
    << endl;

cerr << "qos target peak = " << qos.peak_bandwidth.target << endl
    << "qos target mean = " << qos.mean_bandwidth.target << endl
    << "qos target burst = " << qos.mean_burst.target << endl;

cerr << "qos minimum peak = " << qos.peak_bandwidth.minimum << endl
    << "qos minimum mean = " << qos.mean_bandwidth.minimum << endl
    << "qos minimum burst = " << qos.mean_burst.minimum << endl;

cerr << "Connection Id = " << connectionId << endl;
qos.peak_bandwidth.target = 0;
qos.peak_bandwidth.minimum = 0;
qos.mean_bandwidth.target = 128;
qos.mean_burst.minimum = 2;
qos.mean_burst.target = 1;

status = atm_accept(mainFd, mainFd, connectionId, &qos, dataflow);
if (0 > status)
{
    atm_error("atm_accept()");
    exit(1);
}

#if 0
cerr << "Before fork()" << endl;
status = fork();
switch (status)
{
    case -1: // error
        perror("fork()");
        exit(1);
    case 0: // child
        break;
    default: // parent
        cerr << "Parent exiting." << endl;
        exit(0);
}

cerr << "Child :: After fork()" << endl;
#endif

slaveFd = atm_open(deviceName, O_RDWR, &info);
assert (slaveFd > 0);

status = atm_bind(slaveFd, 0, &serverSap, 10);
if (0 > status)
{
    atm_error("atm_bind()");
    exit(1);
}

status = atm_listen(slaveFd, &connectionId, &calling, &qos, &aal);
if (0 > status)
{
    atm_error("atm_listen()");
    exit(1);
}

cerr << "After atm_listen" << endl;

cerr << "Server SAP assigned=" << serverSap << endl;

strstream myString;
myString << serverSap << ends;
tbuf = myString.str();

status = atm_send(mainFd, tbuf, strlen(tbuf));
delete[] tbuf;

assert(0 < status);

atm_close(mainFd);
cerr << "Child: atm_close(mainFd)" << endl;

// need to create a new
qos.peak_bandwidth.minimum = 0;
cerr << "Child:: before atm_accept()" << endl;

status = atm_accept(slaveFd, mainFd, connectionId, &qos, dataflow);
if (0 > status)
{
    atm_error("atm_accept()");
    exit(1);
}

atm_recv(slaveFd, rbuf, info.mtu-1);
cerr << "Received: " << rbuf << endl;

atm_close(slaveFd);

return 0;
Appendix D

Acronyms

AAL: ATM Adaptation Layer
AAL 3/4: ATM Adaptation Layer 3/4 - Adaptation for Data Services
AAL 5: ATM Adaptation Layer 5 - Adaptation for Data Services
ABR: Available Bit-Rate
ACK: Acknowledgement for TCP packet
ARP: Address Resolution Protocol
ATM: Asynchronous Transfer Mode
ATMARP: Asynchronous Transfer Mode Address Resolution Protocol
BECN: Backwards explicit Error Congestion Notification
BLOBS: Binary Large Objects
B-ISDN: Broadband Integrated Services Digital Network
BSD: Berkeley Software Distribution
BUS: (LAN Emulation) Broadcast and Unknown Server
CAC: Connection Admission Control
CBR: Constant Bit-Rate
CDV: Cell Delay Variance
COE: Context-Of-Execution (a thread or a process)
CORBA: Common Object Request Broker Architecture
CPCS: Common Part Convergence Sub-layer
CRC: Cycle Redundancy Checksum
CSMA/CD: Carrier Sense Multiple Access/Collision Detection
DNS: Domain Name Service
DOC: Distributed Object Computing
ELAN: Emulated Local Area Network
ELF: Executable and Linking Format
FECN: Forwards explicit Error Congestion Notification
FTP: File Transfer Protocol
HDTV: High Definition Television
HTML: Hypertext Markup Language
HTTP: Hypertext Transfer Protocol
IETF: Internet Engineering Task Force
IHL: Internet Header Length
IIOP: Internet Inter-operability Protocol
InATMARP: Asynchronous Transfer Mode Inverse Address Resolution Protocol
IP: Internet Protocol
IPv4: Original Internet Protocol (4 ⇒ 32 bit addresses)
IPv6: Next-Generation Internet Protocol (6 ⇒ 192 bit addresses)
JMAPI: Java Management API
LAN: Local Area Network
LANE: (ATM Forum) Local Area Network Emulation
LEC: LAN Emulation Client
LECS: LAN Emulation Configuration Server
LES: LAN Emulation Server
LIS: Logical IP Subnetwork
MAC: Media Access Control
MPOA: Multiple Protocols over ATM
MSL: Maximum Segment Life. Arbitrarily defined as 120 seconds in [Bra89b]
MTU: Maximum Transmission Unit
NBMA: Non-Blocking Multiple Access
NHRP: Next Hop Resolution Protocol
NHS: Next Hop Server
NIC: Network Interface Card
NNI: Network-Network Interface
OMT: Object Modeling Technique
POSIX: Portable Operating System Interface for UNIX
RFC: Request For Comments
RPC: Remote Procedure Call
RTT: Round-Trip Time
SNMP: Simple Network Management Protocol
SSCOP: Service Specific Connection Oriented Protocol
SSCS: Service Specific Convergence Sub-layer
SSCOP: Signalling ATM Adaptation Layer
SVR4: System V Release 4
SYN: Synchronise sequence numbers flag for TCP packet
FIN: Final packet for TCP connection
T/TCP: TCP for Transactions
TAO: TCP Accelerated Open (for T/TCP)
TCB: TCP Control Block (TCP per-connection state information)
TCP: Transmission Control Protocol
TTL: Time to Live
UCB: University of California at Berkeley
UCB-CSRG: University of California at Berkeley, Computer Systems Research Group
UDP: User Datagram Protocol
UMTS: Universal Mobile Telecommunications System
UNI: User-Network Interface
URL: Uniform Resource Locator
VBR: Variable Bit-Rate
VoD: Video-on-Demand
WWW: World Wide Web
4.4BSD: Final release of Unix-like operation system from BSD.
Bibliography


[Lew96] Bill Lewis. An article on threading... (Usenet cross-posting to newsgroup comp.programming.threads and newsgroup comp.unix.solaris). E-mail: bill@easy1.mediacity.com, Message-Id: <4oagr0$6c2@easy2.mediacity.com>, 25 May, 1996.


BIBLIOGRAPHY


BIBLIOGRAPHY


[Wee96] Computer Weekly. **Review 1996: That was the year that was.** Computer Weekly, 12 December, 1996.

[Won] Clinton Wong. **What’s New in HTTP/1.1 – Design Enhancement on the Road to HTTP-NG.** The WWW Consortium, Massachusetts Institute of Technology, Laboratory for Computer Science, 545 Technology Square, Cambridge, MA 02139, USA.