The Universal Transport System: An Adaptive End-to-End Protocol Analysis and Design

A thesis submitted for the Degree of Doctor
of Philosophy of the University of Technology, Sydney

School of Electrical Engineering,
University of Technology, Sydney

Antony Richards

July 24, 1995

Certificate

I certify that this thesis has not already been submitted for any degree and is not being submitted as part of candidature for any other degree.

I also certify that the thesis has been written by me and that any help that I have received in preparing this thesis, and all its sources used, have been acknowledged in this thesis.

Signature of Candidate

Acknowledgements

Many people have helped to make this research effort a success.

Firstly, I want to thank my supervisor Aruna Seneviratne. He has been instrumental in guiding me through the early reams of the literature review leading into my main focus, and acting as a sounding board. Best of all, he has become a good friend.

I also thank my co-supervisors, Teresa Buczkowska and Michael Fry for their useful suggestions and guidance. I also thank Michael who on that late night HIPPARCH meeting when he came up with a name for my work - *The Universal Transport System*. Again, they have both become good friends.

The University of Technology, Sydney School of Electrical Engineering provided support in many ways. I would particularly like to acknowledge the staff of the Research Computing Centre, Geoff Ingram and Carol Gibson, for providing a place to work, access to SUN Sparc-stations and PCs, and for promptly assisting with any problems. I also thank Geoffi for all the help he gave me with the insides and outsides of UNIX, and for putting up with all my questions.

The help, and many discussions with the RCC residents Tim Aubrey, David Bate-

man, Pranot Boonchai-Apisit, Hyun-Soo Cho, Ranil De Silva, Anne Fladenmuller, Ruben Gonzalez, John Hefferan, Jennifer Jang, Vasantha Saparamadu, Craig Scott, Varuni Witana, and Salim Zaman are greatly appreciated. I would also like to thank Craig and David Lowe for the latex template used by this thesis – this being the fundamental aim of research, being able to use somebody else's wheel instead of inventing one's own.

I thank the Australian Research Council for the financial support provided for the duration of this PhD through an ARC Post-Graduate Research Award. The financial support from Telecom Australia (and in particular Telecom Research Laboratories) is also gratefully appreciated. The discussions that I have had with Chris O'Neill at TRL have also given me invaluable insight into TCP.

The friendships that I made at INRIA (and in particular the people from the RODEO group) during my 2 months exchange are cherished. My family was made to feel at home from the day we arrived.

Finally, I would like to express my appreciation for the support given to me by my parents Ray and Warwick. My wife Nicole has been the greatest support, and did not complain too much about the many late nights I've spent at Uni. Last, but not least, I thank my son Adam for finally sleeping through the night so that I can have a clear head during the day again.

Publication List

1. Ranil De Silva, Laurent Dairaine, Antony Richards, Aruna Seneviratne, and Michael Fry, Automatic generation of dynamically adaptable protocols, in Upper Layer Protocols; Architectures and Applications, December, 1995. submitted.

2. Antony Richards, Ranil De Silva, Anne Fladenmuller, Aruna Seneviratne, and Michael Fry. The performance of configurable protocols. Journal of High Speed Networks, 1995. To be published.

3. Christophe Diot, Isabelle Chrisment, and Antony Richards. Application level framing and automated implementation. In High Performance Networking, 1995.

4. Antony Richards, Ranil De Silva, Anne Fladenmuller, Aruna Seneviratne, and Michael Fry. The application of ILP/ALF to configurable protocols. In HIPPARCH Workshop, December 1994.

5. Antony Richards, Aruna Seneviratne, Michael Fry, and Varuni Witana. Tailoring the transport protocol for giga bit networks. In Australian Telecommunication Networks and Applications Conference, December 1994.

6. Antony Richards, Tamara Ginige, Aruna Seneviratne, Teresa Buczkowska,

and Michael Fry. An adaptive transport service suitable for high speed networks. Concurrency: Practice and Experience, 6(4):357-373, June 1994. ftp://ftp.ee.uts.edu.au/pub/prose/richards.concurrency94.ps.gz.

7. Aruna Seneviratne, Michael Fry, Varuni Witana, Vasantha Saparamadu, and Antony Richards. Quality of service management for distributed multimedia applications. In Phoenix International Communication Conference, March 1994. ftp://ftp.ee.uts.edu.au/pub/prose/seneviratne.phoenix94.ps.gz.

8. Michael Fry, Antony Richards, and Aruna Seneviratne. Framework for implementing the next generation of communication protocols. In Fourth International Workshop on Network and Operating System Support for Digital Audio and Video, 1993. Lancaster, England. ftp://ftp.ee.uts.edu.au/fry.daav93.ps-.gz.

9. Antony Richards, Tamara Ginige, Aruna Seneviratne, Teresa Buczkowska, and Michael Fry. DARTS - a dynamically adaptable transport service suitable for high speed networks. In Proceedings of the Second International Symposium on High Performance Distributed Computing, pages 68-75. IEEE, July 1993.

10. Antony Richards, Aruna Seneviratne, Michael Fry, and Teresa Buczkowska. A case for adaptive protocol implementations. In Australian Broadband Switching and Services Symposium, July 1993. ftp://ftp.ee.uts.edu.au/pub/prose-/richards.absss93.ps.gz.

11. Antony Richards. Darts - a dynamically adaptable protocol suitable for high speed networks. Technical report, University of Technology, Sydney, December 1992. ftp://ftp.ee.uts.edu.au/pub/prose/richards.darts.ps.gz.

Preface

This thesis describes the research undertaken in fulfillment of the requirements for the Degree of Doctor of Philosophy, at the University of Technology, Sydney. This work was undertaken during the period March 1992 to July 1995.

TCP is the network equivalent of DOS – it is quick and dirty, but got there first!
Antony Richards, 1995.

Contents

List of Figures and Tables

Figures

Tables

List of Abbreviations

AAL	ATM Adaptation Layer
ACK	Acknowledgement packet
ADAPTIVE	A Dynamically Assembled Protocol Transformation, Integration, and Validation Environment
ALF	Application Level Framing
API	Application Program Interface
ARP	Address Resolution Protocol
ATM	Asynchronous Transfer Mode
BSD	Berkeley Software Distribution
CISC	Complex Instruction Set Computer
CoRA	Configuration and Resource Allocation
CPU	Central Processor Unit
CRC	Cyclic Redundancy Check
CSMA/CD	Carrier Sense Multiple Access with Collision Detection

Da CaPo	Dynamic Configuration of Protocols
DLD	Dynamic Link/unlink eDitor
DPA	Demand Priority Access
DQDB	Dual Queue Dual Bus
FDDI	Fibre Distributed Data Interface
F-CSS	Function-Based Communication Subsystem
F-PDL	Function-Based Protocol Description Language
FEC	Forward Error Correcting
FTP	File Transfer Protocol
GBN	Go Back N
i486	The 486-33MHz computer
ILP	Integrated Layer Processing
IPC	Inter Processes Communication
IP	Inter-networking Protocol
IPC	Inter-Process Communication
ISO	International Standards Organisation
IVS	INRIA Videoconferencing System
JPEG	Joint Picture Experts Group
LAN	Local Area Network

MAC	Medium Access Control
MAN	Metropolitan Area Network
mips	The Dec 5000/240 computer
OSI	Open System Interconnection
PDU	Protocol Data Unit
PF	Protocol Function
QoS	Quality of Service
RISC	Reduced Instruction Set Computer
RTP	Real-Time Transport Protocol
RTT	Round Trip Time
Rx	Receive or Receiver
SDL	Specification Description Language
SE	Synthesis Engine
SLIP	Serial Line IP
SML	Session Management Layer
SR	Selective Repeat
SunOS	The Sun (computer) Operating System
TCP	Transmission Control Protocol
TDT	Theoretical Departure Time

TFTP	Trivial File Transfer Protocol
TML	Transport Management Layer
TP0	Transport Protocol 0 (part of the OSI stack)
TP4	Transport Protocol 4 (part of the OSI stack)
TPDU	Transport Protocol Data Unit
Tx	Transmit or Transmitter
UDP	Unreliable Datagram Protocol
UTS	Universal Transport System
UX	the UNIX server running under Mach
VC	Virtual Circuit
VP	Virtual Path
WAN	Wide Area Network
WBC	Wide Band Channel
XTP	Xpress Transfer Protocol

Abstract

This thesis presents a study that investigates the operation of end-to-end protocols and shows the benefits of a general purpose adaptive approach. The application of this work includes protocols for high speed networks, multimedia networks and mobile networks where the QoS of the underlying data links and/or the requirements of the application are variable.

The study is composed of a number of steps. Firstly several end-to-end protocols are decomposed into their atomic protocol functions. Compound protocol functions are then synthesised by combining the atomic ones. A mapping from the various application classes into a set of protocol functions is also developed.

Experiments are then performed that quantify the processing overheads of TCP that are associated with the error control and flow control protocol functions. Two platforms are compared, a 486 and a Dec 5000/240. Both platforms use Mach 3.0. The results show that protocol customisation resulted in improved performance.

The need for an adaptive protocol is then outlined. The implementation of adaptivity is then investigated by modifying TCP so that various protocol functions can be selected during a data transfer. This modification uses dynamic linking (DLD) to allow the protocol to be very flexible. The results show that adaptivity introduces only a small overhead.

A generic adaptive framework is then presented that uses the atomic protocol functions to implement a general purpose protocol. This framework supports protocol implementation concepts such as ILP and ALF.

Chapter 1

Introduction

The rapid growth of the Internet highlights the growing need to network computers so that resources can be pooled. The underlying technology is evolving from lower speed wired networks such as Ethernet, to include mobile links, and high speed connections (for example ATM). At the same time, the use of networks are changing from traditional applications like file transfers and distributed file systems to include interactive multimedia programs.

The current end-to-end computer protocols, such as OSI TP0, OSI TP4, and TCP have rigid structures, and inappropriate functionality for many of these emerging combinations of network topologies and applications. Further, many studies have shown that protocol processing (especially above layer 3 of the OSI Reference Model) represents a major bottleneck in current computer designs. This bottleneck is expected to become even more dominant in future computer systems.

The dominant property in the field of computers is the rapid rate of change. However the rate of change is not evenly spread. Thus, new bottlenecks are being created, and must be addressed.

The following sections will outline the current trends in this field, show the need for change, and then outline the general approaches to a solution.

1.1 Network Evolution

1.1.1 Legacy Local Area Networks

Ethernet

Ethernet is the most widespread LAN in use today. As shown by figure 1.1 it consists of a common bus. Each host operates independently of the others. The maximum bandwidth available is 10Mbps, with a packet size that allows up to 1500 bytes of user data and protocol overheads. LAN segments are limited to 2.5km [69].

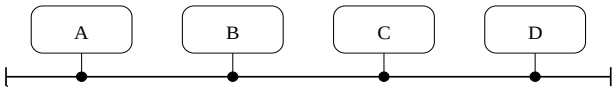


Figure 1.1: *An Ethernet Network.*

When a station transmits data it must first listen to the network to see if it is already in use. If the network is idle then the station may start transmitting data. As all the stations on the Ethernet are independent, another station may have started transmitting data at the same time. This is known as a collision.

To ensure reliability, transmitting stations must detect collisions. This is achieved by the transmitter also listening to the network. When the transmitted data and received data differ, a collision has been detected. The host will then stop the data transmission, wait for a random delay, and then start the transmission process again. This scheme is known as Carrier Sense Multiple Access with Collision Detection (CSMA/CD).

Due to the effects of collisions, the throughput of an overloaded Ethernet tends to zero.

Token Ring

Token rings are also used in the LAN environment. As shown in figure 1.2, the network consists of a loop with stations connected to it. When a station wants to transmit some data it must first wait for a token to arrive. Once a station has the token, it is allowed to transmit data onto the ring for a pre-determined period of time. This data circulates around the ring until it returns to the original station where it is removed. The token is then passed onto the next station.

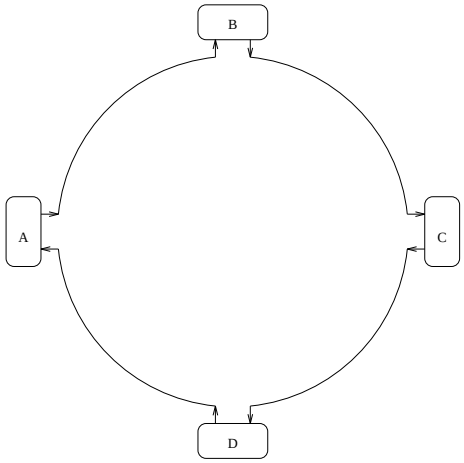


Figure 1.2: *A Token Ring Network.*

This scheme guarantees a minimum bandwidth to each station, even under overload conditions. A typical maximum bandwidth of a token ring is 16Mbps [40].

1.1.2 High Speed Networks

The current goal in high speed networking at the network layer (layer 3) is to achieve transmission rates at 100's of Mbps and at the Gbps, while being able to support both real time and non-real time services. These will need to be provided in the local, metropolitan and wide area networking environments.

Several solutions are currently available which provide connectivity at several Mbps. The current technology, which is based on fibre optics, also results in very low bit error rates.

Switched Ethernet

Switched (or Fast) Ethernet has a similar operation to, and uses the same packet format as normal Ethernet except that it can support data speeds up to 100Mbps. LAN segments are limited to 210m [69].

Demand Priority Access

Demand Priority Access (DPA) is a LAN designed to operate over both fibre optics (with a maximum length of 2km) and twisted pair cables (with a maximum length of 100m). Speeds up to 100Mbps are supported. Both Ethernet and Token Ring frame formats are recognised.

As shown by figure 1.3, the topography of the LAN is based on hierarchy a (or tree) of the repeaters. The top node is known as the root repeater. The leaf nodes represents the stations connected to the network.

Two levels of priority are supported in this topography, normal priority and high priority. A stations gain access to the network by sending a request to the attached

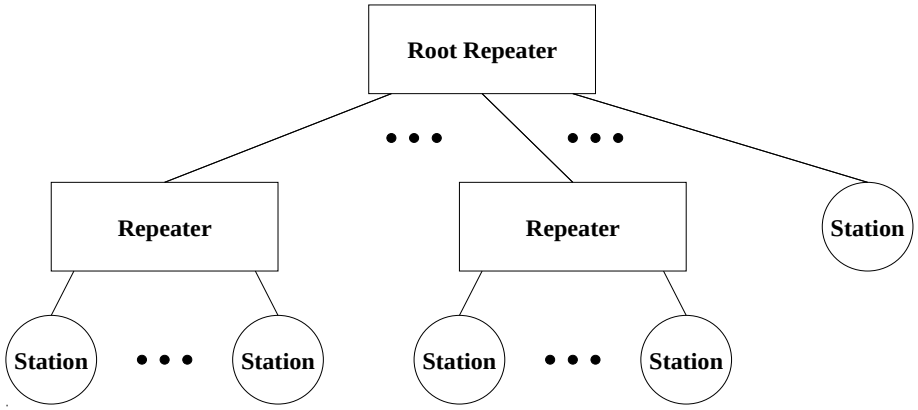


Figure 1.3: *A Multi-Level DPA LAN. Taken from [69].*

repeater. The repeater services the high priority requests first. Requests with the same priority are serviced using a round robin arbitration strategy.

If the request requires access to a higher (upstream) repeater, then the attached repeater will propagate the request through. Once the attached router has been granted permission to transmit then all outstanding normal priority requests in the repeater are transmitted, except that this may be interrupted by a high priority request.

To ensure proper operation of the protocols and fairness within the LAN, whenever a normal priority request is blocked for longer than about 250ms it becomes a high-priority request.

FDDI

FDDI (Fibre Distributed Data Interface) has developed from the token ring approach. For reliability, FDDI has two physical rings. Thus if the cable is cut at any point, then the two stations adjacent to the break can loop the data through the backup ring, as shown in figure 1.4.

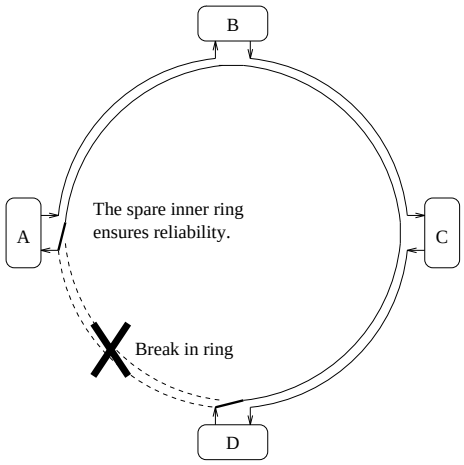


Figure 1.4: *The Use of the Spare Ring to Ensure Reliability when the Ring is Cut.*

The network can be configured so that under normal operations either the spare ring is idle or can be configured to carry traffic.

FDDI is based on fibre optics, with network speeds up to 100 Mbps and distances less than 50km. The token ring scheme used has provision for priority levels and each station must guarantee the maximum delay that it will hold the token. This gives an upper bound for the access time to the ring. Thus FDDI can support real time (as well as non-real time) data streams.

FDDI-II, an extension of FDDI, supports synchronous traffic by subdividing the available bandwidth with up to 16 full-duplex Wide Band Channels (WBC). Each WBC has a bandwidth of 6.144 Mbps. A station may sub-allocate this channel into multiples of 8 Kbps. The idle WBCs are used to form the packet channel (which has a minimum bandwidth of 768 Kbps). Within the packet channel, there is also a priority scheme [69].

DQDB

Another network technology which operates up to hundreds of mega bits per second is known as DQDB (Dual Queue Dual Bus) and has been adopted by the IEEE 802.6 working group as a MAN (Metropolitan Area Network) standard. As shown by figure 1.5, this network consists of two cables, each transferring data in opposite directions.

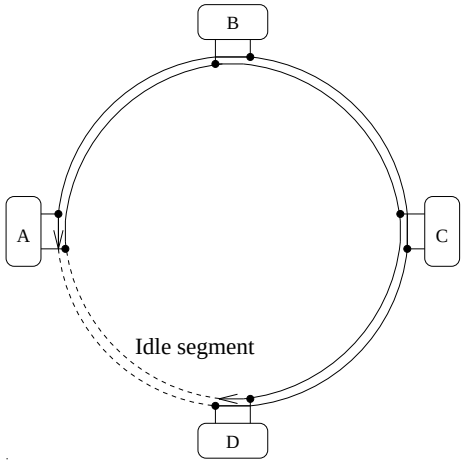


Figure 1.5: *The Topography of a DQDB Network.*

Although DQDB does not require a ring topography, physically it is installed as one. Under normal operation one segment[1] of the physical ring is idle. However, if a segment of the network is cut, the network is reconfigured so that idle segment corresponds with the cut segment.

Through this, DQDB has improved efficiency over FDDI (token rings), even under erroneous conditions [69].

[1] A segment is used to denote a link between two adjacent nodes.

1.1.3 ATM – Unifying LANs and WANs

Asynchronous Transfer Mode (ATM) is a high speed networking approach that aims to integrate both asynchronous and non asynchronous traffic into one physical network. The network is based on packet (cell) switching principles. Each cell is a fixed size and consists of 5 bytes of ATM header with 48 bytes of data.

As shown by figure 1.6, a physical network is superimposed with a set of virtual paths (VP). These are used to create virtual networks. End-to-end connections through the network, known as virtual circuits (VC), are made by using the virtual paths. Different classes of services (such as voice and data) will use different virtual paths even if the different data sources travel from a common source to common destination.

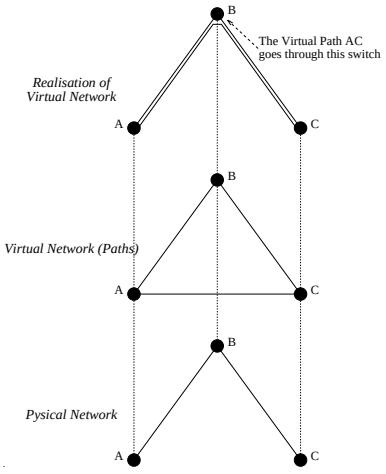


Figure 1.6: *The Relationship between a Physical ATM network, and its Virtual Paths.*

Traffic control is performed to ensure that the traffic in one VP (or VC) does not adversely affect traffic in the rest of the system. That is the VPs (and VCs) are isolated from each other. A traffic controller monitors the statistics of the data (such as the average and peak bandwidth consumed) and if pre-defined thresholds are exceeded cells are either discarded or marked as low priority.

The ATM Adaptation Layer (AAL) [51] interfaces ATM with the higher protocol layers. Four classes of services are defined, labelled classes A to D. There are four AALs specified to satisfy these four classes.

Class A provides fixed bit rate circuit emulation, such as in a traditional telephone network. The negotiated bandwidth can vary from a few Kbps to tens of Mbps. The AAL (1) must smooth out the variable delay introduced by ATM. When cells are lost, dummy cells may be used to fill in the gaps.

Class B provides variable bit rate service with time synchronisation between the sender and receiver. It is envisaged that real time services such as compressed voice and video would use this class.

Class C is for connection oriented data services. That is, the AAL (3/4) must establish an ATM connection (either point to point, or multipoint). The connection may be assured (that is with error control and flow control), or non-assured (that is cells may go missing, or have bit errors present). Further, the messages that are passed to/from AAL 3/4 are defined to be either fixed or variable.

Class D is for connectionless data services. Although the data services are connectionless, ATM itself is connection oriented. Connectionless frames[2] are sent over predefined ATM connections. (Datagram cells are impractical due to the small payload within a cell). A CRC is calculated over the whole frame.

1.1.4 Wireless Networks

Wireless LANs have recently become economically viable, and are currently being standardised by the IEEE (802.11). At present there are several products on the

[2] A Frame denotes the block of data that enters the AAL for this class.

market that are designed to be plugged into Laptop PCs.

The characteristics of a wireless based network are vastly different from that of a wired one. They generally have lower transmission rates, Kbps to a few Mbps, and vastly different error characteristics. Further, the available bandwidth varies over time. Commercial products that operate at 2Mbps with a range up to 330m are readily available [8].

There are several modes of operation envisaged for these LANs. A wireless network may have a bridge to the cable LAN in use (at present this is usually an Ethernet based network). Equipment may be stationary or roaming. The latter involves the mobile equipment being passed from one base station to another, in a similar manner to cellular telephones.

Finally situations will exist where peer to peer mobile communications will be required. For example two (or more) machines used during a conference or meeting may need to communicate only to each other.

It can be seen that there are varied uses of mobile networks, thus the services provided will also be varied.

1.2 Host System Evolution

The diversity of the computers that will be connected to these networks is also increasing. They will range from PCs and workstations with varying capacities, to supercomputers and mobile devices such as personal digital assistants.

In the case of PCs and workstations, the power of the central processor is improving at a steady pace. However, the speed of the machine's memory is not improving at the same rate as the central processor. At present, the main memory of a com-

puter operates between 4 to 20 times slower than the CPU [82]. To allow the CPU to operate at a speed unrestricted by the memory speed a high speed memory cache is used.

RISC processors are becoming increasingly popular over the traditional CISC machines. Although these processors allow the CPU to perform faster processing, the difference between the speed of the CPU and main memory is even greater than that of CISC machines. Thus RISC machines become very reliant on an adequate cache.

A consequence of this is the movement of data must be minimised. Further, the data touching/manipulating operations must be grouped together so that the user data is not repeatably loaded into the memory cache.

Mobile devices have to deal with additional constraints, such as its weight must be limited [36]. The main weight associated with mobile equipment is the battery. Thus, conserving the power consumption of the equipment is vital. This requirement has led to compromises in the equipment's design. For example, screen lighting improves readability (for example contrast in some models improves from 6:1 to 13:1), but due to its power requirements some portable products do not light the screen.

Other design decisions that are a function of the power consumed include the speed of processor. Thus, it can be expected that some mobile computers will have a lower processing power than desk-top machines.

A communication interface also consumes power. As pointed out by [36], the cost (in terms of power) to transmit data is about ten times that of receiving it. The bandwidth available to mobile equipment is also limited both due to power consumption and due to the limited electro-magnetic frequencies available.

Thus, as will be shown in chapter 2, the functionality of end-to-end protocols will be different in this environment.

1.3 Network Application Evolution

The current generation of networked applications are of two forms. Either they perform some form of file transfer (such as FTP [67]) or they are modelled on a client-server design. In either case, a client sends a request to a server. The server performs this request, and then replies. The client, which has been waiting for the reply, continues. These applications have loose time restrictions, that is, there are limits to the time taken by these applications, but these limits are not very stringent. Further, the bandwidths required by these applications are relatively small.

Future networked applications have different requirements. Figure 1.7 illustrates an example of such an application - a video conference involving the image of each participant, an audio link and a common white board. These applications will be interactive. The different media (image, sound etc.) will have different requirements with respect to error characteristics, throughput demands, acceptable delays, acceptable delay jitter and so on. For example, the audio link is more sensitive to loss and jitter than the video links, however, the video link requires greater bandwidth.

Another networked application is control systems and manufacturing which requires strict real time delays and guaranteed delivery from the communication system. Networked remote procedure calls need low latency delays, and will tend to generate short bursts of traffic [74].

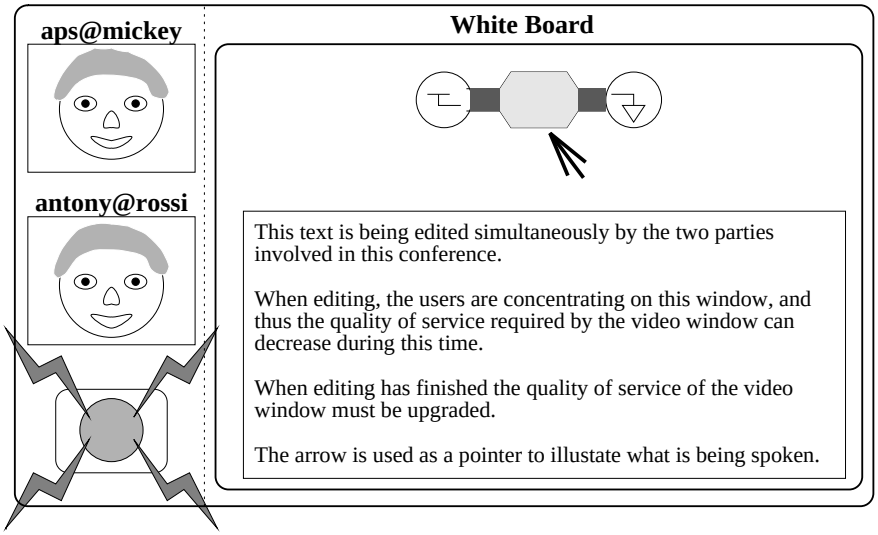


Figure 1.7: *A sample Multi-media Workstation Desktop.*

1.4 Consequences for Current Protocols

It can be seen from the above that protocols need to cater for a wide range of environments. Physical networks are available with a large range of bandwidths. Further, the end systems operate with a large range of processor powers.

To effectively support this, the information transmitted will need to be scalable. A coarse gain realisation of this would involve choosing to transmit only parts of a multimedia document. For example the moving images may sometimes be removed from the document. Alternatively a still image may be used to substitute for the moving one. A fine grain realisation of this approach would be to change the resolution of an image [63].

The delays and error characteristics introduced by a network also vary. Thus a selection of error detection and correction mechanisms are needed.

The current trend in processor design implies that the memory cache utilisation is important. Thus Integrated Layer Processing (ILP) [21] based designs are beneficial.

Finally, multimedia applications will be able to accept errors in the data stream. The acceptable maximum error rate will vary over time. Further the resources consumed by an application will also vary over time.

Five illustrations of how the current protocols are inadequate are presented.

Application Specific Requirements

In many instances only the user will be able to determine a suitable response to adapt the use of the communication system [61]. For example video may be transmitted over a network to either play back at the other end, or to perform video editing.

Both applications communicate using the same data and data structures. However in the case of limited bandwidth the adaptations should be different. The video player must maintain a real time playback at the cost of maintaining the original frame rate, resolution and image quality. The video editor on the other hand must ensure that every frame is seen (to allow precise editing) at the cost of real time playback [61].

Thus the communication system has to make decisions that are dependent on the user.

Audio/Video Transmission

Whenever data is lost or corrupted, the current protocols stop the data flow. A retransmission scheme is then used (Go Back N or Selective Repeat). This introduces a round trip delay into the data flow, which translates into a noticeable pause in the audio and video. If the loss is frequent, this causes the audio to be unintelligible. However, if no retransmission schemes are used, then lost packets will result in

shorter pauses, and improved audio quality. Techniques also exist, such as inserting white noise, that minimise the perceived effect of missing packets.

Similarly, for video, improved quality results from not retransmitting lost data. Lost packets can be replaced by replaying the previous image.

Variability of Requirements

Another observation from figure 1.7 is that the requirements of the application (known as its quality of service, QoS, requirements) vary over time. For example, at the beginning of a multimedia conference, the participants will greet each other. Attention is focused on the pictures of the participants. Then, as they get to work, attention will focus on the white board. This allows the quality of the participant's images to be degraded.

Synchronisation

Synchronisation involves co-ordinating the timing or two or more independent data streams. The term *Lip Synchronisation* originally applied to synchronising audio with the movement of a person's lips. The term has now been generalised to include synchronising different media such as audio, video, pointers, graphics/images, animation and text.

If the data is out of synchronisation then human perception tends to identify the presentation as artificial, strange or annoying. However the perception of synchronisation is very dependent on the types of media that are related. It has been shown that related data streams that have a small degree of skew may still appear to be synchronised [78].

Multimedia networks must deal with synchronisation issues because it is often not practical (or possible) to multiplex all the media into one stream (for example each media will have different QoS requirements). Thus the receiver must perform re-synchronisation.

Subjective experiments performed in [78] showed that humans considered a video/audio clip of a news reader to be synchronised if the skew between the two media was less than ±80ms. The presentation was considered out of synchronisation if the skew was greater than ±160ms. The viewers could also tolerate a level of skew without being distracted (that is the skew was detected but it was not considered annoying). This tolerance was greater when the video was ahead of the audio[3].

Other experiments performed in [78] showed that when a pointer was required to be synchronised with audio (a discussion) then a skew up to ±250ms was considered acceptable. A skew above ±1500ms was not acceptable. Between these values the skew was noticed, but could be tolerated.

Finally it can be seen that the permissible skew between different media varies with the application. The range of permissible skew vary from ±5ms for audio with a tightly coupled image, to $-500/+750$ms for audio with a pointer[4] [78].

Thus synchronisation will need to be included into the protocol stack used by some applications, and not included in the stack used by other applications.

[3] It is probable that the greater tolerance for video being ahead of audio is because it also occurs as a normal experience. A distant person talking will produce this skew because light travels faster than sound.

[4] The range specified is that the audio is behind by 500ms to the audio being ahead by 750ms.

Mobile Networks

Work by Cáceres [11] has shown that flow control within TCP degrades the performance of a connection through inappropriate functionality. An example of this problem is when a mobile unit moves from one (radio) cell to another. When the cells are non-overlapping, packets will get lost. This is due to routing inconsistencies during a hand-off, which can last up to 0.15s [11].

However, the flow control algorithms within TCP [43] interprets this loss as congestion. Thus, TCP's response, after the expiry of a timer, is to reduce its transmission bandwidth. This results in a further drop in the overall throughput, with a typical end-to-end data flow pause of 0.8s.

Cáceres also argues that due to the increased error rates of mobile networks, a selective repeat error control scheme should be used in preference to the Go Back N scheme used by TCP.

Discussion

It can be seen from the above, and further analysis in chapter 2, that the current suite of end-to-end protocols are not adequate in terms of their timing constraints, the bandwidth available at the top of the protocol stack (compared to the raw bandwidth available), the functionality provided by the protocols and the rigid user interface. Thus, it can be seen that the current generation of protocols need to be modified so that they will be appropriate for future applications.

1.5 Partial Solutions

The sections above have illustrated the unsuitability of the current protocols because of their inappropriate functionality. This has been addressed by a large number of researchers. The solutions range from modifying the implementation of a protocol without changing its functionality to developing new protocols. This section will give an overview of the proposals and as shown by chapter 2, that none of the schemes will provide a complete answer to the problem.

Implementation Optimisations

The identification, and consequent optimisation of the common path of a protocol has been very successful in improving efficiency [85, 91]. The improvement is for two reasons, firstly the efforts of a programmer are focused where it is required, and secondly cache utilisation is improved because, once on the common path, there are no further conditional statements.

Another optimisation that does not affect the external operation of a protocol is known as Integrated Layer Processing (ILP) [21]. This optimisation improves the memory cache utilisation by requiring all per word operations to be performed in a single loop.

However, improving the efficiency of the current generation of protocols will not solve all of the problems identified in §1.4, such as networked audio and video. The functionality of future protocols also need to be modified.

New Protocol Mechanisms

New protocol mechanisms have been proposed to improve protocol functionality. An example is known as Application Level Framing (ALF) [21]. The concept is that the packet used to transfer data is one that is meaningful to the application. This allows packets to be processed out of order.

The Xpress Transfer Protocol (XTP), a new generation general purpose protocol, has had several several new protocol mechanisms introduced into it. These include having a specific field in its header to allow rate control, and separating the control and data packets by requiring the transmitter to periodically request the current state of the receiver [83].

Custom Protocols

New protocols have been designed to improve the efficiency of a protocol for a given application (and host environment). These protocols are lightweight in that they do not perform all the functionality of a general purpose protocol. Examples are VMTP [15] (designed for request-response applications) and NETBLT [19] (designed for bulk data transfers).

Part of these protocol's design is that operations have been grouped together so that they are performed only once per group of packets rather than for every packet. This, combined with new protocol functions that will be discussed latter, has resulted in improved efficiency by reducing the level of control data being transferred and by simplifying the flow of control.

However, being custom protocols, they are not suitable for general purpose use.

Protocols Tailored to an Application

This chapter has illustrated the diversity of underlying networks and emerging applications. In order to match this large range of variables, while not experiencing a high processing overhead, tailored protocols are being advocated. Two approaches are being proposed to support such protocols.

The first approach is to automate the generation of a protocol per application by using compilation techniques [28]. This requires the application to be parsed so that the required protocol functions are determined. The output of the parser is then compiled to produce the protocol implementation.

This approach customises the protocol during the compilation of the application. Thus the protocol will not take into account issues such as the current network load and the end system resources. Further, unless specified, this approach will not allow the protocol to support adaptation.

The second approach (which is the main topic of this thesis) is to develop a generic framework that allows a protocol to be synthesised at run time [72, 74]. The use, or non-use, of the specific protocol functions is decided during the data transfer.

Both the protocol compilation approach and the synthesised protocol approach require a mapping from the application's requirements into the available protocol functions. In order to achieve this the protocol needs to be decomposed into its component functions [73, 93], and then a mapping needs to be developed from the application's requirements into the protocol functions.

1.6 Generic Solution

In order to provide an effective communication system for future applications and networks an integrated solution needs to be developed. The solution shown by figure 1.8, represents a generic end-to-end adaptive protocol architecture.

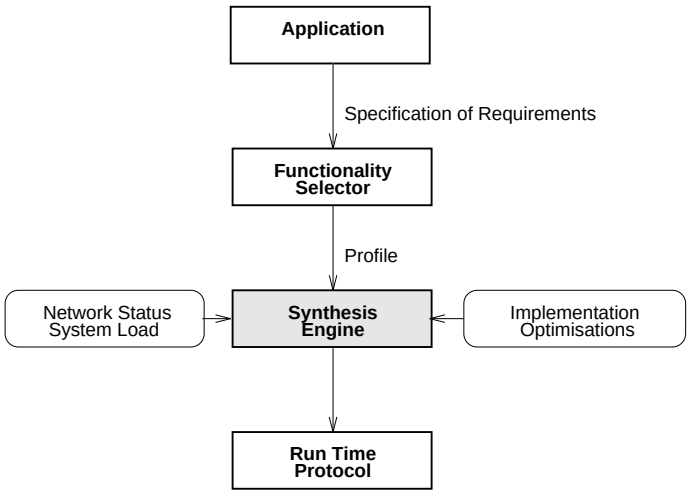


Figure 1.8: *The Conceptual Architecture of the Proposed Adaptive Architecture.*

A specification of the application's requirements (also known as its QoS requirements) is provided by the application. The specification includes the application's preferred degradation path (for example colour to black and white). This specification may change during the life of the connection.

The application's specification is then translated by the Functionality Selector into a set of protocol functions (known as a Profile) that are required to support the application's needs. The protocol functions are specified in terms of error control, flow control, etc.

The Synthesis Engine uses the profile to create a run time protocol. The protocol will be affected by the status of the network, the current system load, and implementation optimisations (which will be outlined in the next chapter).

To support this concept the operating system must provide resource reservation and management. The communication system must provide adaptive profiles. Current systems (such as the UNIX operating system and the Internet) do not support these notions [34].

The system represented by figure 1.8 addresses these problems by attempting configuration and re-configuration of the system's various components.

This thesis is primarily concerned with the operation of the Synthesis Engine.

1.7 Specifying Protocol Requirements

A formal specification is required to map the specification of the application's requirements into the operation of the communication protocol used. The Application Requirements Specification has been defined in [27] as a three-tuple consisting of ordering, reliability and timing. Figure 1.9 illustrates this.

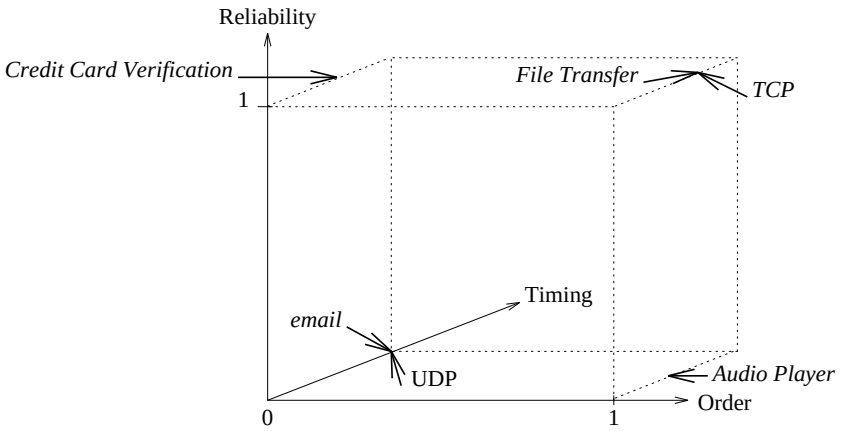


Figure 1.9: *The Three Axes of QoS Required by Multimedia Applications. The requirements of some current applications are also illustrated (in italics) for the reliability-order axes. The service provided by TCP and UDP are also indicated.*

The ordering constraint specifies whether the data must be delivered to the ap-

plication in order, in partial order, or whether out of order delivery is permitted. The reliability constraint specifies the guarantee that transmitted data will eventually arrive at the destination. The timing constraint specifies the minimum and maximum acceptable transmission delays [27].

A number of schemes exist that allow these requirements to be specified. Methods include using language techniques such as SDL (Specification Description Language) [54] or petri-nets [4]. Figure 1.10 gives an example of a petri-net of the JPEG player developed in [28].

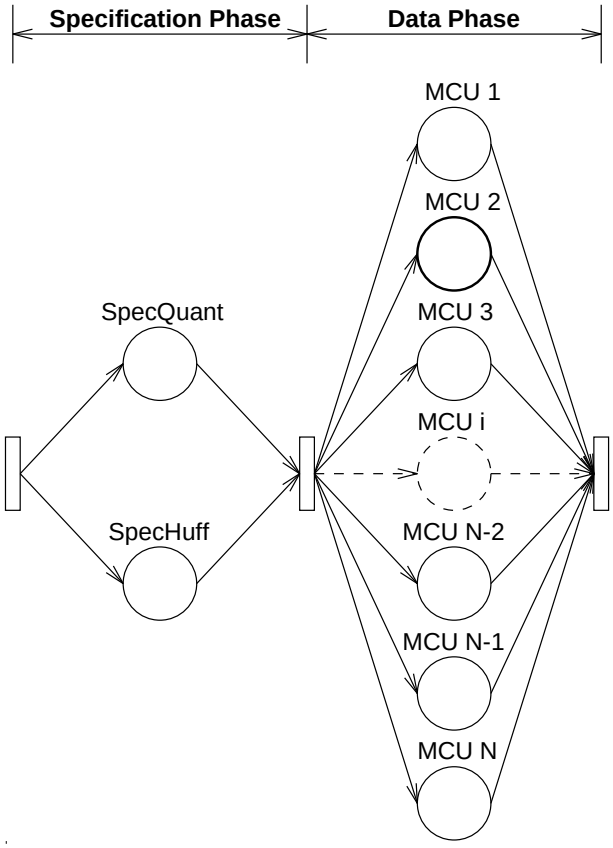


Figure 1.10: *A Petri-net Specifying the Order Requirements of a JPEG Player.*

The transmission of a image consists of two phases. The first is to transfer the control data for the image. The control data can be accepted at the receiver in any

order. However, to end the specification phase, all the control data must be received. The second phase is the data transfer. During this phase the data packets may be received in any order. The transfer is completed when all the data is received. This petri-net has not defined any timing constraints.

The focus of this thesis is in the design and implementation of flexible protocol stacks. It does not investigate how an application will specify its protocol requirements.

1.8 The Proposal in this Thesis

A protocol can be considered to be composed of a set of protocol functions that are configured to satisfy a specific set of requirements [12, 35]. The functions may be configured by compilation, or at run time to synthesis the protocol.

The language approach, such as Esterel [5] and SDL [54], is capable of specifying protocol requirements and creating efficient customised protocol code is being developed. Thus, an application programmer can specify the application's requirements, which is translated into a Petri-Net model, and then used to create an custom protocol during the compilation stage of the application.

The other approach is to configure a protocol server during the communication session. This may be done by putting conditional statements into the protocol software, or more systematically by combining specific protocol functions. Further, dynamic linking [90] can be used to allow greater flexibility and extendibility when synthesising the protocol.

However, some problems do exist with this approach. For example, a synthesised protocol does not naturally allow ILP techniques to be applied. Infact quite the

opposite is true, by separating the the protocol functions, the data manipulation functions will also be separated. These problems will be further investigated in chapter 5.

Although application specific protocols have been advocated [12, 35, 84], to our knowledge, a systematic study of the overheads associated with designing and implementing these protocols have not occurred.

It is asserted that both performance gains and necessary functionality of the protocol can be achieved by tailoring protocols to the application's needs when using either of these approaches. In order to test the validity of this assertion it is necessary:-

- to determine the potential sources of overhead in the implementation of a protocol suite,
- to show that protocol mechanisms can be factored out to achieve performance gains, and
- to synthesise "appropriate" protocols by combining protocol functions.

This dissertation reports results of a study carried out to determine the possibility of implementing flexible, application specific protocols and then presents the design of a general purpose adaptive protocol, known as the Universal Transport System (UTS).

1.9 Organisation of this Thesis

Chapter 2 discusses the related literature. Chapter 3 provides a methodology for decomposing end-to-end protocols into their component protocol functions, and then

reports about experiments using a protocol derived from TCP/IP to see the effects decomposition has on its performance. These experiments also illustrate the operation and bottlenecks of TCP/IP. The next chapter, chapter 4, examines the operation of an adaptive implementation of a TCP derivative. A design of a general purpose adaptive protocol and its associated packet structures is presented in chapter 5.

Finally, the thesis concludes (in chapter 6) that adaptive protocols are both desirable and feasible.

Chapter 2

Analysis of Current Network Protocols

The older network protocols (TCP/IP and OSI TP4) have been designed for networks with expensive and/or limited bandwidths compared to the available processor power of today. Further the bandwidth-delay product within older networks tended to be small. Thus the design optimisations of these protocols included [29]:-

- the liberal use of processor power to reduce the transmission costs,
- the use of only relatively simple flow control mechanisms due to the small bandwidth-delay product.

However, due to the evolution of the underlying network technology and processor designs, the design optimisations of future network protocols are different. Further, current literature shows that communication bottlenecks are developing within the end-to-end protocols [21, 29, 49].

A range of views is being put forward for dealing with these bottlenecks. Literature indicates that the problems are caused by the host system architecture, the operating system support, the protocol mechanisms and/or their implementation.

Watson [88] has performed an extensive analysis of the issues that need to be considered for a transport service by comparing the operation of lightweight protocols with that of general purpose protocols. The study shows that lightweight protocols achieve improved throughput. This improvement in performance can be attributed to the following three basic reasons [29, 88]:-

1. the use of use better implementation techniques,
2. the elimination of unneeded functionality, and
3. the use of improved protocol mechanisms to achieve a given functionality.

To achieve the above, lightweight protocols tend to be tied to a particular application or network architecture. Given that these are evolving rapidly, together with the problems with porting software, the conclusion is that it is better to optimise the general purpose protocols based on the lessons learned from the lightweight implementations. In addition, Watson [88] concludes that, if the three techniques were applied to general purpose protocols, it will be possible to achieve similar performance to the lightweight implementations.

Thus, the proposed solutions to allow general purpose protocols to support high speeds, range from restructuring the computer's operating system to suit the requirements of protocols, investigations into protocol implementation methodologies, to the implementation of totally new protocol mechanisms and architectures.

2.1 Implementation Techniques

Clark [21] also argues that the inefficiencies in the current protocol stacks are due to the implementation methodologies as well as the protocol mechanisms being used. Both Clark [21], and Crowcroft [23] argues that it is the layering concept itself that leads to inefficient protocol implementation. This is because a layered protocol model tends to produce optimal solutions for each layer, rather than producing an overall optimal system. They also highlighted other issues such as the number of context switches, the number of data copying operations and the excessive use of timers that will also cause inefficiencies.

Doeringer [29] points out that due to the very low error rates of high speed networks, protocol implementations must optimise the protocol processing for the error free (normal) case. Further, the protocol will need efficient operating system support.

To quantify the processing overheads of end-to-end protocols, the performance of TCP/IP will be illustrated. After this, the various implementation optimisations available will be presented.

2.1.1 A Queueing Model of TCP/IP

A study of the overheads of TCP/IP in the SUN-OS (version 4.0.3) environment was performed by Papadopoulos [65, 66]. In this study a queueing model for TCP/IP was developed and used to instrument the software.

The model involves three queues at both the transmitter and receiver. The first queue at the transmitter is operates at the user/kernel interface. It connects the application buffer to the socket buffer. The second server represents the operations

of TCP/IP. It connects the socket buffer to the device driver interface queue. Finally the device driver interface queue is served by the device driver, which outputs data to the Ethernet.

Data enters the receiver at the device driver and is placed on an IP queue. The result of the TCP/IP processing moves this data into the socket buffer. Finally, data leave this queue for the application via a system call.

Table 2.1 summarises the measured throughput reported in [66]. On the Sparc 1 with continuous data flow, the average service time required by TCP/IP processing at the sender was 370μs, compared to 260μs at the receiver (this included the check-sum calculation in both cases). For a comparison, the time required to perform a copy of 1024 bytes was 130μs.

	Sparc 2	**Sparc 1**
SPECint rating [26]	21.7	about 12.8
Throughput (Mbps)	8.5	7

Table 2.1: *The Throughput of TCP/IP Achieved by Papadopoulos.*

The queue feeding the network card was fairly large during continuous operation. This indicates that the network layer is the main bottleneck of the system. The reason given in [66] is that consecutive Ethernet packets cannot be transmitted back-to-back.

2.1.2 Non Data Touching Operations

Kay [46] has observed the overhead required by protocol operations (including the relevant operating system support) using fine grain measurements. It was observed that for existing Ethernet usage, the average TCP message was 32 bytes, while the average UDP message was 128 bytes. Over 99% of TCP messages were less than 200

bytes, compared to 86% of UDP messages. (Nine percent of UDP messages were 8192 bytes long).

When this is taken into account Kay has shown that it is not the data touching operations that limit performance (which is the case for large message sizes). The performance is limited by the non data touching operations. For TCP with 32 byte packets only 16% of the total processing time is consumed by the data touching operations.

The non data touching operations can be placed into five groups [46]:-

Protocol Specific Processing: Operations such as setting packet headers and maintaining state information.

Network Buffer Manipulation: Operations on the `mbuf` data structure.

Other Data Structure Manipulations: This includes socket buffer and interface queue manipulations.

Operating System Functions: Support for sockets and synchronisation.

Error Checking: Performing parameter checks on system calls.

It can be seen that these operations include operating system support for network services.

For the average sized packets using Ethernet at present, the protocol specific operations represent the highest overhead, but this overhead is not dominant. Thus to significantly improve the performance of the communication system requires improvements in all five types of operations.

2.1.3 Projected Performance of TCP/IP

Analysis of the receiver side of TCP by Clark [18], which is summarised by table 2.2, shows that on a 32 bit, 10 MIPS RISC chip, the control part of TCP over FDDI could handle a throughput up to 530 Mbps. However, the per byte operations will limit the throughput to 32Mbps (this assumes 250ns, 32 bit RAM with four memory cycles per input packet). Based on 70ns RAM[1], this limit becomes 114Mbps.

Instructions	Operation/Function
57	IP
154	TCP receiver common path
15	Sequencing & buffer management (Data packets only)
17	Processing window field (Data packets only)
9	Processing ACK (Control packets only)
20	Processing window (Control packets only)
17	Outgoing congestion window calculation (Control packets only)
44*	RTT calculation (Control packets only)
35	buffer layer receiving data
30	buffer layer receiving an ACK

Table 2.2: *The Receiver Instruction Count for TCP/IP on a 386 Processor. These instruction counts are calculated, and confirmed by observation in [18] on a version of TCP/IP in isolation of the operating system – such a realisation is not realistic. Also, these figures do not include optimisations such as TCP header predictions.*
** The RTT (Round Trip Time) is calculated once per round trip, not each packet. For Berkeley UNIX, there is an upper-limit of 22 instructions per packet.*

The implementation used includes some (but not all) of the implementation optimisations that will presented in this chapter. Thus it can be seen that when properly implemented, the control part of TCP protocol processing is capable of handling the throughput requirements of high speed networks.

[1] The Sparc 2, 4 & 5 and the faster 486 machines use 70ns RAM.

2.1.4 Memory Cache Usage – Integrated Layer Processing

Protocol processing can be divided into two parts, control functions and data manipulation functions. In the data manipulation part, the actual data (or pay-load) of a PDU is read from memory, manipulated and possibly returned back to memory. Examples of data manipulation functions include presentation encoding, checksumming, encryption and compression. In the control part there are functions for header and connection state processing. Clark et al. have demonstrated that the control part processing can match gigabit network performance for the most common size of PDUs with appropriate implementation [18]. However, data manipulation functions present a bottleneck [21, 39]. Data manipulation consists of two or three phases.

The time to read and write to memory dominates the processing time for many protocol functions, The checksum is such a function. For other protocol functions, the data manipulation time dominates. Examples of this include the encryption function and some presentation encodings.

The time to read/write data is dominated by the speed of the computer's memory. A computer's main memory operates from 4 to 20 times slower than the CPU [82]. High speed (and thus expensive) memory is used as a cache to work around the inefficiencies due to the slow memory. A cache operates by duplicating the "active" parts of memory.

This method is effective if, and only if, the memory usage pattern of the program being executed is correlated. Current protocol implementations do not optimally use a cache. This is because the data field has to be manipulated several times, as already outlined. These functions are spread over different layers. In a naive protocol suite implementation, the layers are mapped into distinct software or hardware entities which can be seen as atomic units. The functions of each layer are carried out completely before the protocol data unit is passed to the next layer. This means that

the optimisation of each layer has to be done separately. Such ordering constraints is in conflict with efficient implementation of data manipulation functions [23].

Thus, a naive protocol implementation requires the protocol's pay-load to be loaded into the cache (from the slow speed memory) for the first data manipulation function. Once the data item has been touched by the current atomic unit, it will not be touched again for a long time. Thus it is probable that the data will not be in the cache for the next data manipulation function.

It would be suitable for an optimal cache manager to release the memory associated with the protocol's pay-load the next time data needs to be loaded into the cache. However, real cache managers predict future use patterns by ordering the cache by the last time a memory location was addressed. This approach implies that a real cache manager will differ from an optimal one, in that the data will remain in the cache for a short time.

The net effect of this is that not only will the protocol be inefficient because the data pay-load will repeatably need to be loaded from main (slow) memory, but the execution of the protocol itself will be slowed down because the effective size of the cache is reduced by the amount of "stale" data present. Thus the program instructions and control data will need to be loaded from the main memory into the cache more often than required had the pay-load been accessed only once.

An implementation principle called Integrated Layer Processing (ILP) [21, 39] has been proposed to address these problem. With integration it means that several layers are implemented within the same module. This does not mean that the implementation is unstructured. The basic idea behind ILP is to perform all the manipulation steps in one or two processing loops, instead of performing them serially as is done today. A loop involves a read from memory, followed by all manipulations and a write back.

Thus there is a reduction in the number of memory accesses. As this discussion has shown, it is expected that ILP will lead to better a performance improvement on RISC processors compared to CISC processors.

To facilitate ILP a protocol architecture should be organised in such a way that the interactions between the control and data manipulation functions do not interfere with their integration. For example, an error control function may result in the packet being discarded, but if the operation was integrated with copy the data into user space then the user cannot receive that data despite it already existing in its memory space.

Recent work [9] has shown that adjacent loops may be sufficient for efficient implementation of complex functions.

2.1.5 Application Level Framing

Application Level Framing (ALF) involves segmenting data into messages that are meaningful to the application [21]. This allows out of order processing of the data because all the relevant information is contained within each message. This concept will also facilitate ILP because it will allow messages to be processed in isolation from other messages.

Experiments have shown that ALF will result in performance gains whenever there is a reasonable probability that messages will arrive out of sequence [28].

2.1.6 Application Level Protocols

Another approach to minimise the cost of protocol processing is presented in [57, 84], where the bulk of the protocol processing is performed by the application process.

The main argument for this approach is that one less protection domain boundary is crossed, and thus there will be an improvement in efficiency.

Experiments by Maeda involved placing the protocol processing required during the data flow stage of a connection (that is not during connection establishment or closing) in user data space [57]. (To implement this the kernel required a packet filter that had access to the header of the arriving packets). This eliminates the system-user space data copy and associated operating system overheads such as context switches. On a Dec 5000/200, a throughput of 8 Mbps was achieved when using the Mach operating system. This is comparable to an in-kernel approach, and substantially better than having a separate process to perform the protocol operations [57].

However, some operating systems, such as Mach [7], provide *copy on write* IPC buffers between applications. If an application never writes over the IPC data, then the only extra over-head is a context switch[2] and the processing required to set up the IPC buffer. This overhead is required only once per block of data passed between processes. Shared memory would eliminate the *copy on write* overhead if the application is guaranteed not to modify the data, or if control of the data is relinquished. Blocking IPC calls can be used to guarantee synchronisation between processes.

The application level processing approach also has a problem when a *fork()*[3] system call is encountered. The system call causes a new process to be created, with all the variables copied. Thus, the open connections will have two points of execution (the parent and child).

There have been three solutions proposed to solve the above problem. The first

[2] The user level protocol approach would still need a thread switch.
[3] The *fork()* system call in UNIX creates a new process.

one is to pass the open connections to a central server. Thus the protocol's operation is no longer in the application's data space. The second approach is not to use *fork()*, but just create new threads as required. This has the problem of having globally visible data. The third solution is to tell *fork()* who gets which connection. This requires all existing networked software to be modified if it is to access the user protocol libraries.

Another problem is that many protocols, such as TCP, need to maintain their state after the connection (and therefore after the application has terminated) [84]. Under normal conditions it would be possible for the application to hand the connection over to a central protocol server. However, if the application has an abnormal termination, such as a program fault, the hand-over may not occur. This results in a decrease in the reliability of the whole network.

However, the user level protocol design does allow a simpler resource allocation and priority scheme. This is because the destination application, and thus the purpose of the incoming data is known very early in its execution path. In turn, the scheduler and the resource allocator can make decisions for the incoming packets based on the destination's requirements.

This information is not available to kernel or server based implementations until after demultiplexing is performed (at the transport layer). However, quality of service field(s) can be added to the network header of the packets (such as the *type of service* field in IP). This field would then be trusted until demultiplexing where the true values would be known. Further, the intermediate nodes would also be able to adjust its resource allocation schemes and priority levels to that required by the end applications. This scheme relies on applications being honest when setting these fields.

The priority levels used follow the packets through the execution path. For the

user level approach, a single thread of control would have the same priority level. The kernel/server approach requires threads to change their priority level. If self-modifying code [58] is used, this represents a minimal increase in overhead.

2.1.7 Optimising the Common Path by Prediction

The protocol by-pass concept [91] optimises the performance of a protocol by finding the common path through it, and then using optimised code only for this path. A test is made to see if the current packet satisfies the conditions to by-pass the normal protocol implementation, and if so the optimised code is used. This approach does not alter the (external) functionality of the protocol.

At the receiver additional state variables are required to enable a prediction of the next incoming packet [85, 91]. A test is made whenever a packet arrives at a host to see if it is the expected one (performed using a binary comparison of two array - `bcmp()` in UNIX). If this test is true, then the optimal path is followed.

For a good performance improvement to be gained, a large percentage of the packets need to be processed by the optimised common path. Performance results in [91] indicate that an improvement of 120% to 75% can be gained for OSI TP0 (a low functionality OSI transport protocol).

2.1.8 Implement the Protocols in Hardware

Chesson (XTP) [16], Siegel et al. [76] and Zitterbart [93] have proposed hardware based implementations of the transport layer. The advantage of this method is that it is possible to implement general purpose protocols. However, complex (and thus expensive) silicon implementation designs are required because of the complexity of

the protocol mechanisms.

A silicon implementation also tends to produce protocols with ill suited functionality. Further, it has been shown that it is not the protocol processing itself that will limit the throughput of transport protocols, but their interaction with the host operating system. Special purpose hardware still needs to interact with the host operating system, and the general computing environment. Thus the operating system will require similar processing overheads whether it performs the protocol operations or if it implements an interface to the hardware [18].

2.1.9 Upcalls

Clark [17] has proposed a method of structuring and scheduling a protocol, using a system called *Upcalls*. An Upcall is a subroutine call from one module or layer of the software to another. This strategy reduces the number of context switches encountered when processing a protocol because the transitions between the various modules become direct and does not require any scheduling decisions with its associated overheads.

X-kernel, using Upcalls and a buffer management strategy which dramatically reduces the amount of data copying required, has demonstrated that it is possible to obtain significant reductions in protocol processing overhead [42].

2.1.10 Discussion

Although it has been shown by Clark [18] that the control part of TCP is capable of supporting high speeds, and many successful implementation optimisations exist, it has been shown that this is not enough. The functionality of the protocol must be

appropriate for the application's needs. Further, experiments involving an optimised TCP running on a DEC 240 utilised only 84% (8.4 Mbps) of an Ethernet, but only 21% (21 Mbps) of a FDDI. However, the processor was not 100% utilised during these experiments [37].

Thus, although implementation optimisation is important, it is not sufficient to permit high speed data communications. The choice of the protocol functions used, and their realisation must also be revised.

2.2 Elimination of Unnecessary Functionality

As shown by the previous section, the use of various protocol functions results in performance penalties. This can be an unjustified overhead if the application does not even require the functions supported, for example providing order for networked JPEG player given in §1.7.

The use of unnecessary functionality will impact on performance in two ways. Firstly, there is a direct penalty resulting from the associated processing overhead. The checksum calculation in TCP is such an example [71].

The other penalty results from the protocol functionality itself. For example, providing data in order will cause out of sequence messages to be delayed until the missing message is received. Thus the out of sequence message must be queued. This halts its processing (even if the processor is idle while waiting for the next message). When a delayed message arrives the processor then must fully process this message as well as any out of sequence ones. The average time available for this processing will be the same as that available for a single message. Thus, although providing sequenced data does not represent a significant increase in the level of processing, it still degrades the performance of the protocol due to the scheduling

of the processing requirements. Further, an unnecessary delay is introduced to data that is out of sequence.

By customising a protocol to the needs of an application, the performance of the communication system can be improved. A protocol can be customised to an application in several ways. In particular, user level protocols [84] allow each application to have its own protocol designed and implemented. Automated techniques are being developed to simplify the development of these protocols. These techniques allow the protocol to be specified in a high level language, and then compiled into the protocol [1]. The performance of such a protocol is as good as hand-optimised application specific software [28].

Another systematic approach to designing application specific protocols has been proposed by O'Malley [62]. This approach is to design a protocol graph for each application. The graph is composed of protocol functions (referred to as micro-protocols in [62]) and virtual protocols. A virtual protocol represents the control actions and decision points of the protocol graph. The protocol graph (and thus the protocol used) is configurable through the use of these virtual protocols. However, the configuration is not arbitrary as the protocol graph itself is static, just the path through the graph can vary.

2.3 New Protocol Mechanisms

Newer transport protocols such as VMTP [15] and NETBLT [19] have been designed for specific applications that utilise high speed networks. The differences between the protocols stem from the optimisations carried out to tailor them for the application: NETBLT for bulk data transfers over long delay paths (such as satellites) and VMTP for request-response applications (such as a distributed operating system).

Apart from that, the main differences between the specialised and the standard protocols are in the buffering techniques, flow and error control, and timer and connection management [38]. The new protocol mechanisms should be optimised for high bandwidth, low delay and low error rates [29].

Table 2.3 summarises the major differences between the newer protocols with the standard ones. The table shows firstly that there are only a limited number of protocol functions, namely connection management, lifetime control, error detection and correction, sequencing, flow control and multiplexing. The differences between the numerous protocols arise as a result of (a) the use of different algorithms to implement these functions and (b) due to implementation optimisations.

Protocol	Connection Set-up	Error Control	Flow Control	Comments
TCP	Hand-shake based.	Checksum on TPDU. 1 by 1 ACK. GBN window. Timer at Tx.	1 by 1 ACK. Coupled with error control.	Tx must timeout for lost packets. Has a heavy reliance on timers.
OSI/TP4	Hand-shake based.	Length Field of packet. Checksum over TPDU Optional. 1 by 1 ACK. GBN window. Timer at Tx.	1 by 1 ACK. Coupled with error control.	
NETBLT	Modified hand-shaking arrangement	State info sent when a block is received. Receiver based timeouts, using inter-arrival times. SR window. Allows CTS transmision with errors. Inter-arrival timer at Rx.	Uses Window & rate modulation. Tx and Rx buffers are allocated & negotiated for each frame. RTD dependent.	If the Window is larger than the buffers then rate modulated required to reduce prob. of buffer overflow. Runs on top of IP.
VMTP	Timer Based	Response to requests acts as an implicit ACK. One Tx timer per request. SR window.	Based on message groups. Stop & wait at buffer level. Assumes lost packets from queue overflows, not bit errors.	Assumes best effort datagram sub-network (eg IP). Geared for REQUEST / RESPONSE type applications. Designed as a simple base for higher level connection control.
XTP 3.2	Implicit	Two Checksums (header and user data). Uses Poll/Status, and always generates NAKs when data lost.	Separate Flow control window. Also has rate control.	All fields are 32 bit aligned. The design allows great flexibility.

This table is based on information primarily from [15, 30, 38, 83].

Table 2.3: *A Comparison of Different Transport Protocols.*
Note: CTS – Continuous; RTD – Round Trip Delay.

Secondly, specialised protocols contain a subset of these mechanisms chosen according to the requirements of a given application and network service.

This section will describe the differences between error control, flow control and timer support with the various protocols.

2.3.1 Error Control

Errors in end-to-end protocols are caused by damaged, lost, duplicated and mis-sequenced packets. Timers and checksums are used in conjunction with a retransmission scheme to detect and correct errors. In earlier networks, a common cause of errors was the bit errors occurring in the underlying network. Correction was achieved by retransmitting the corrupted packet.

The mechanisms need to be different for high speed networks because of their low bit error rates and high bandwidth-delay product. In these networks, if a packet is lost, it is most likely due to a buffer overflow. If retransmission of the packet is required (for non real-time applications), it must be done within the bandwidth allocated to the communication channel, otherwise the congestion will be increased further. This scheme requires an agreed data rate to be negotiated during connection establishment, which can be re-negotiated at a later stage.

A common error control scheme (used by TCP and OSI/TP4) is to use a sliding window in conjunction with an acknowledgement scheme. At the transmitter, a retransmission timer is set whenever a packet is sent. The value of the retransmission timer is normally greater than the time taken to transfer a packet and receive its corresponding acknowledgement. If the transmitter has not received an acknowledgement on expiry of the retransmission timer, it will retransmit the packet. This is very process intensive, as timers frequently need to be set and reset[4]. Further-

[4]TCP times a single message per RTT is timed. Thus only one active timer is required per session. However this scheme only works for GBN retransmission schemes because the most recently transmitted message will not be acknowledged unless all preceding messages have been received.

more, it is difficult to determine an appropriate time out value. It depends on the system load, system resource scheduling characteristics and round trip delay. Long time out periods will drastically affect the throughput of erroneous connections. A short time out period will lead to unnecessary retransmissions.

Both NETBLT and VMTP have reduced the number of timers required by grouping the packets, and acknowledging the reception of these groups. NETBLT measures the inter-packet arrival time at the receiver because this is less variable than the round trip delay time measured by TCP and OSI. It also allows the timers to be placed at the receiver, which knows exactly which packets have arrived.

XTP only requires one timer because it has separated the data and control packets. This timer is used to initiate a request for the current state of the receiver (known as a poll packet).

2.3.2 Flow Control

Flow control bounds the amount of unacknowledged data that can be outstanding within a communication session. This is so that there is a bound on the maximum amount of end system resources used by a connection.

A sliding window flow control mechanism operates by limiting the number of unacknowledged packets at the transmitter to a fixed number. In order to guarantee that the receiver cannot be flooded with data, the size of the window must be less than the maximum amount of buffering available.

However this scheme does not work well whenever a packet is buffered at the receiver (for example the receiving process may not be ready to accept more data).

The message with the lowest sequence number is retransmitted whenever a timeout occurs.

The reason is because if an acknowledgement is generated immediately then the transmitter is free to send more data. However if the acknowledgement is delayed then the transmitter's timer may expire resulting in unnecessary retransmission of the data. A modification to the sliding window scheme, known as a credit scheme, addresses this problem. When the receiver acknowledges data it also explicitly specifies how much further data (credits) it is willing to accept. This allows the window size to change dynamically. When used correctly, this results in improved performance over a straight sliding window protocol. This scheme is used by TCP, TP4, and XTP.

Within the common protocols (such as TCP and OSI TP4), flow control shares a common window with error control. However, from their definitions, it can be seen that they represent very different functions. Further, as shown by Clark et al. [19], when these two functions are coupled it may result in very poor network performance. For example, when packet retransmissions takes place, it takes place outside the flow control's window, and thus increases congestion.

Most transport protocols do not explicitly address network congestion. However, general purpose transport protocol implementations can attempt to avoid it. In one proposal [43], the protocol implementation observes a principle of *conservation of packets* by adjustment of its behaviour in response to timeouts.

In addition, a particular protocol implementation will have its own operational requirements. For example, to reach steady state, an algorithm known as slow start is often used[5]. For it to be effective, data loss must be primarily caused by congestion rather than due to random bit errors. The algorithm's operation is as follows:-

- an initial credit (known as a congestion window) of one is set, and the initial

[5]Infact the slow start algorithm is part of the TCP standard.

threshold is made large,

- if the congestion window is less than the threshold, it is increased by one for each acknowledged packet received,
- if the congestion window is greater than or equal to the threshold, then the congestion window is increased by one each round trip time.
- when a timeout occurs or loss is detected, the threshold is set to half the current window size.
- when loss is detected, the congestion window is halved,
- when sending data, send the maximum of the receiver's advertised window and the congestion window [22, 43].

If the receiver advertises available buffer space in bytes as soon as it is available, it may lead to the *silly window syndrome* occurring. The behaviour associated with this syndrome is that the receiver's window oscillates between zero and a small value. Although the sender may have a large queue of data waiting, it transmits small (non-full) packets to fill the receiver's window as soon as it opens. This behaviour results in poor network utilisation.

To avoid this from occurring within TCP, the receiver will delay opening a window (from zero) until it can advertise a non-trivial window size. Non-trivial means at least a maximum segment size, or a quarter of the buffer size [22].

Another problem with flow control results from applications sending a byte stream to the communication session (such as from `Telnet`). If these bytes are transmitted immediately, it results in many very small TCP packets in the network, each with at least 40 bytes of TCP/IP header. This leads to unnecessary network congestion.

Nagle's Algorithm [59] has solved this problem by allowing only one unacknowledged small packet. For many applications (such as `Telnet`) the byte stream is echoed, causing an immediate return packet. This packet also acknowledges the first one. Thus, the byte stream will be converted into a series of packets, one per round trip time. If the application does not echo the received data, then as a result of delayed acknowledgements used by TCP, the return acknowledgement will be delayed a further 200ms – thus one packet will be sent roughly every 200ms.

This algorithm has only one weakness, illustrated by escape codes within a `Telnet` session. An escape code consists of two 8 byte characters, the escape character followed by another. Without Nagle's algorithm in operation, the bytes are sent immediately. However, with Nagle's algorithm, the escape character is sent immediately, but the second byte will have to wait for the acknowledgement. However, as the receiver's application is waiting for the character after the escape character, it does not echo anything. Thus, the character will be delayed for a round trip time plus 200ms (for a LAN this is effectively 200ms). In other words, if a user types an escape code over a LAN `Telnet` session, there will be a 200ms delay before a response is seen - a perceivable delay[6] [80].

NETBLT uses a flow control mechanism referred to by Watson as a blast protocol [88]. The basic algorithm for this protocol is that a complete buffer is transmitted (possibly with rate control). Transmission is then suspended until a single acknowledgement is returned. In the case of NETBLT, at the beginning of each buffer transmission the rate of transmission between the transmitter and receiver is negotiated. This allows either end to control the rate of data transfer.

If a sliding window scheme has a very large window, and uses delayed acknowl-

[6] A corollary of this is that `Telnet` session should use the `TCP-NODELAY` option for LAN connections.

edgements, then it is effectively the same as the blast protocol. That is, the blast protocol can be seen as a special case of a sliding window protocol.

VMTP has flow control implemented at the packet group level. A packet group cannot be transmitted until the previous group has been acknowledged. This scheme provides efficient flow control if (and only if) the sender is slightly faster than the receiver.

In another proposal [68] the transport layer uses explicit feedback from the network layer in order to regulate network congestion. In both of the above, the action taken to alleviate congestion is to modify the window size, which effectively controls the rate of transmission.

2.3.3 Rate Control

Rate control limits the rate that data is transmitted. Rate control is needed for networks with intermediate nodes because, if these nodes are congested, then data will be lost. Flow control only directly addresses the available buffering of the two end nodes and not the intermediate ones. However, it is common to limit the flow control to achieve rate control (for example slow start within TCP). This is effective as long as there is no significant loss due to bit errors.

True rate control limits the number of packets per second that are allowed to be sent. One scheme to implement this is to specify the maximum rate allowed, and then use a timer at the transmitter to enforce this. However this scheme does not allow for variance in the rate of data generation.

TCP does not use explicit rate control, while NETBLT implements a periodic rate control scheme. The stop and wait protocol (at the packet group level) allows VMTP to perform rate control.

Finally, XTP has a `rate` and `burst` field within its header which can be set be the end systems and the intermediate nodes. A timer at the transmitter is set to expire after $\frac{burst}{rate}$ units of time. That is the timed period times the `rate` gives the maximum allowed burst (in bytes).

The implementation of XTP's rate control scheme requires the use of another state variable, known as the `credit`. At the start of each timed period
credit is set to `burst` and is decremented by the amount of data transmitted. If it reaches zero then transmission is halted [83].

Monitoring the Rate

A scheme, known as a leaky bucket, allows streams with variable instantaneous rates to be monitored. The monitoring works on the traffic's long term mean. The operation of this scheme is given by figure 2.1.

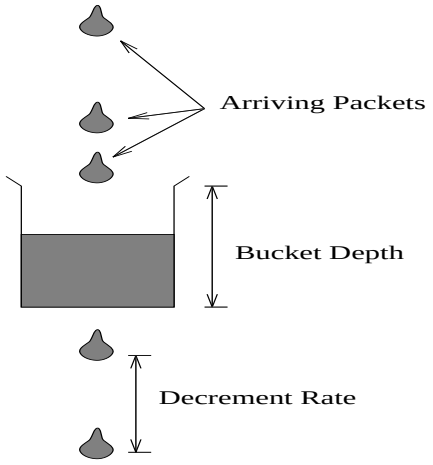


Figure 2.1: *The Conceptual Operation of a Leaky Bucket. If the bucket overflows, then packets are delayed.*

Whenever a packet is transmitted, the counter within the bucket is incremented. If this value is below full, then the packet is allowed to be sent. If the leaky bucket is performing shaping, then the packet is delayed until the counter is decremented.

Otherwise, the leaky bucket is performing traffic control, and the packet is discarded or is tagged as low priority. Decrementing of the counter occurs at a periodic rate.

The depth and rate of decrementing determines the properties of the traffic. If the depth is large, then then rate of decrementing the leaky bucket corresponds to the maximum long term average rate of traffic. However, if the depth is small, then the rate corresponds to the maximum peak rate of the traffic. Leaky buckets can be combined to provide greater control of the traffic.

Another mechanism that performs rate control is shown by figure 2.2 was derived from [10]. This scheme does not use a timer, instead it must look up the current system time. This is achieved by using the Theoretical Departure Time (TDT).

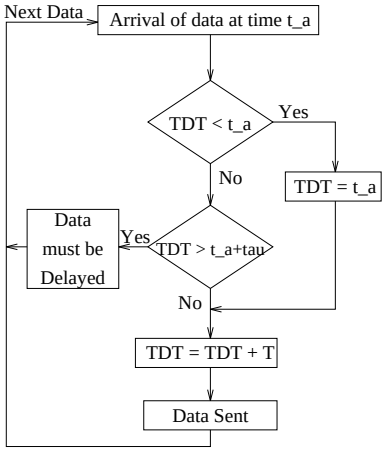


Figure 2.2: *The Conceptual Operation of the Virtual Scheduling Algorithm. TDT = Theoretical Departure Time*

When data is to be sent, the current time (t_a) is compared with its TDT. If the current time is less than the TDT, then the TDT is set to the current time and the data is sent. If the TDT is greater than the arrival time plus τ (a constant that allows fluctuations in the data rate) then it must be delayed, otherwise it is sent. If the data is sent, then TDT is incremented by the average time between departures (T).

This scheme can detect long term violations in the rate of data transfer.

2.3.4 Connection Establishment

A scheme is required to prevent false connections due to delayed and duplicated control packets. Two schemes are in use. A three way handshake is used by TCP and OSI TP4. An implicit scheme is used by XTP.

The packet exchanges involved in a three way handshake is shown in figure 2.3. Under normal operation, a connection request is sent from machine A to machine B. Machine B replies with a connection accepted packet. When machine A receives the accept packet it replies with a confirm packet.

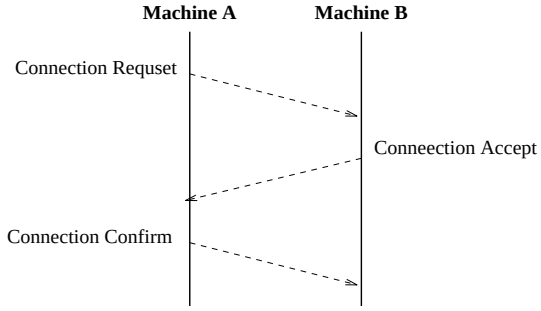


Figure 2.3: *The Packet Exchange Involved in a Three Way Handshake.*

To ensure reliability, a timer is used at each end. If machine A does not receive an expected connection confirm in time (due to the request or the accept packets being lost or delayed), it will retransmit its connection request. Likewise, if machine B does not receive an expected connection confirm in time (because either the the connection accept or connection confirm packets was lost or delayed) it will retransmit the connection accept packet.

Figure 2.4 shows the effect of a delayed connection request. Machine A retransmits the request after a period of time. The connection establishment then continues

as normal. After a long period of time, the delayed connection request reappears. Machine B does not known that this was a delayed packet, so it responds to this connection request with a connection accept packet. When this packet reaches machine A it is rejected because the machine has not attempted to establish a connection. For reliability, the initiating machine cannot create a second connection with the same destination address and port until it can be sure that all previous control packets no longer exist in the network. This is achieved by having a `Time To Live` field in the packet.

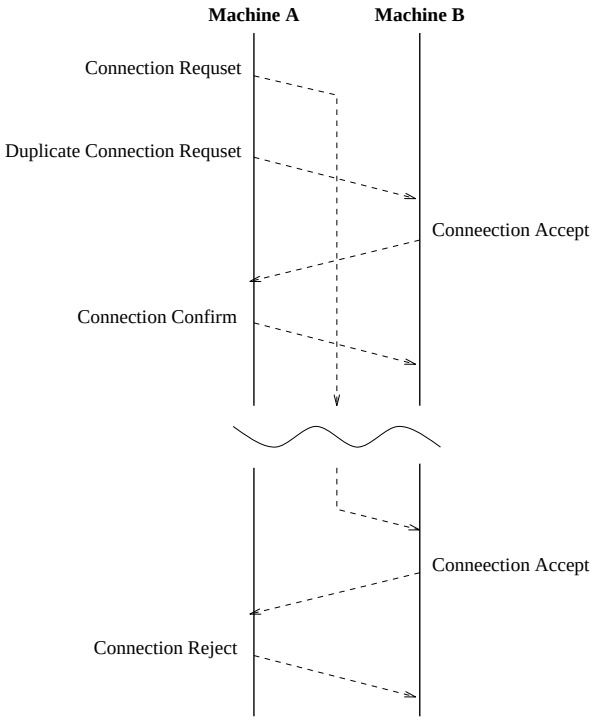


Figure 2.4: *The Effect of a Delayed Packet.*

The three way handshake connection establishment requires a round trip time delayed between the initial request to establish a connection and the first instant that data may be transfered. However, if QoS parameters need to be negotiated between the two ends, it would be possible to piggy-back this onto the handshake.

To overcome this, an implicit connection establishment scheme is used by XTP.

In this scheme, the initiator of a connection just sends the initial data assuming that the receiver will accept it. Once the first packet is received at machine B, it is compared to any of the active connection to test for duplicate packets. If required, a return acknowledgement will packet confirm the connection. A unique key in the packet structure is used to ensure reliability against delayed and duplicated packets. This key cannot be re-used until it can be guaranteed that all previous packets with the same key cannot be in the network. Similarly to TCP/IP, this is achieved by using a `Time To Live` field within the packet structure.

2.3.5 Timers

Timers are essential for a protocol's operation over an unreliable network. However, the number of timers used must be limited for several reasons [3]. Firstly, timers represent a high processing overhead for a protocol. This is because timers are used to monitor many events. Thus a large number of timers are required. This problem is compounded because most operating systems did not consider the needs of data communications when the timer operations were optimised. VMTP has reduced the need for timers by only monitoring groups of packets.

Secondly, as argued by Zhang [92], a timeout only indicates to the protocol that something is wrong. A guess, based on incomplete information, is required to determine the cause of the timeout. The decision, which is based on a guess, cannot be optimal, and thus the response is non optimal. For congested networks, the current use of timers (to cause a packet to be retransmitted) will typically exacerbate the situation.

The variance of the interval to be timed also reflects on the protocol's overhead. If the interval is highly variable, then it must be over-estimated. This leads to a slow

response to errors. However, if the timed interval is reduced, then false time-outs will occur. NETBLT measures the packet inter-arrival time at the receiver instead of the normal measurement of the round trip delay time for this reason. This requires only a single timer at the receiver.

XTP requires only one timer, to initiate the poll packets at the sender. The value of this timer determines the rate that resources are freed at the end systems, and is independent of the round trip time.

2.4 Conclusion

It can be seen that there are many optimisation techniques available to improve the performance of end-to-end protocols. These range from improving the implementation of existing protocols, to eliminating protocol functionality, and to developing new protocol functions. The best solution will be a combination of all these approaches.

In particular customised protocols will need to be developed. However, it should be noted that all the protocols studied in this chapter are composed of a limited number of functions and mechanisms. This should make it possible to tailor a protocol to the requirements of an application.

The next chapter will present a systematic approach to developing a tailored protocol by factorising out the functions used in the current generation of communication systems. The resulting performance from this factorisation will then be examined.

Chapter 3

Protocol Operations

In order to improve the performance and tailor the functionality of communication systems to an application, it is necessary to determine how protocols can be decomposed and recombined. The most logical approach would be to decompose the communication subsystem according to functionality, that is in terms of connection establishment, error control, flow control, etc.

This chapter first presents an end-to-end protocol decomposition. It then presents the results of experiments carried out to access the effects of such a decomposition on the execution performance of the most widely used and analysed transport protocol, TCP.

3.1 Protocol Decomposition

As shown by table 3.1 all end-to-end protocols consist of a common set of protocol functions. Thus it should be possible to synthesise any protocol by recombining the

required protocol functions.

	Connection Establishment	Error Control	Error Detection	Flow Control	Congestion Management
UDP			√		
TCP	√	√	√	√	√
XTP	√	√	√	√	√
VMTP	√	√	√	√	√
NETBLT	√	√	√	√	√

Table 3.1: *Analysis of End-to-end Protocol Functionality.*
Key:
√*- This function is present.*

In order to develop a factorised model of end-to-end communication systems a factorisation of the protocol functions is required. For this work, a protocol function is defined to be a set of realisations (referred to as protocol mechanisms) that have something in common. For example the protocol mechanisms Parity and Cyclic Redundancy Check (CRC) can both implement the Error Control protocol function.

Protocol functions have intricate inter-dependencies. Thus they cannot be removed in isolation. By decomposing a protocol in a "systematic" fashion, these inter-dependencies can be determined. The various Protocol Mechanisms available were also identified. The decomposition that was created is given in Figure 3.1. This model was derived through investigations of both the OSI TP4 standards [13, 14], the TCP/IP documentation [70], realisations of the protocols [22, 79] and related literature [30].

The lines in the figure represent dependencies of one protocol function on another. For example, Data Flow depends on Segmentation & Reassembly, Ordered Delivery, Connection Management, Error Control, rate Control and Flow Control being present in the protocol.

In turn, both Segmentation & Reassembly and Ordered Delivery use the Buffer

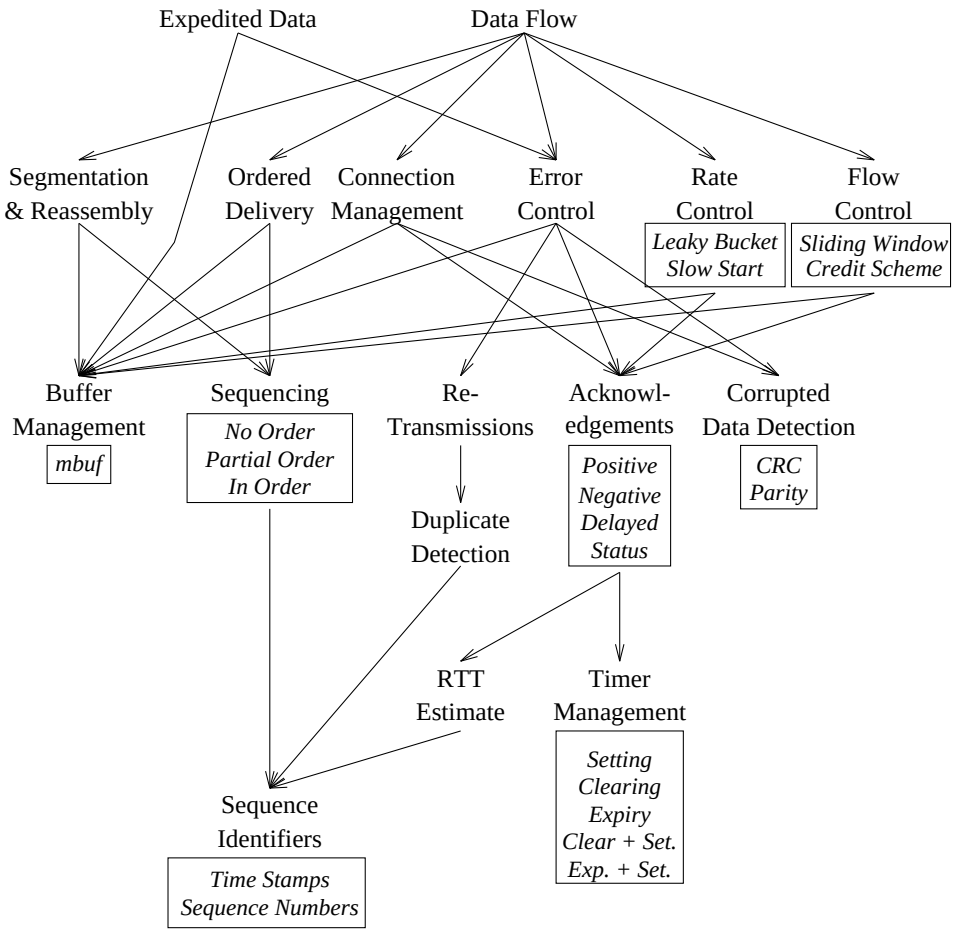


Figure 3.1: *A Dependency Graph of Transport Protocol Functions.*

Management and Sequencing functions. Connection Management is supported by Buffer Management (connections cannot be made without allocating memory), Acknowledgements (to ensure that a connection is still alive), and Corrupted Data Detection (if a high percent of the data is corrupted then the connection is not effective). Error Control requires Buffer Management (to store packets for retransmission), Retransmissions, Acknowledgements (so that successful transmission can be deduced) and Corrupted Data Detection (to decide if a retransmission is required). Rate Control is supported by Buffer Management and Acknowledgements. Possible realisations of Rate Control (as shown by the box on the figure) are Leaky

Buckets and the Slow Start algorithms. Flow Control requires Buffer Management (to delay the transmission of data), and Acknowledgements (to adjust the window). Possible realisations of Flow Control include Sliding Window and Credit Schemes.

At the third layer, Buffer Management can be realised using mbufs. Likewise, Sequencing, which depends on Sequence Identifiers, can be realised using No Order, Partial Order or In Order. If Retransmissions exist in the communication system, then Duplicate Detection is required. Duplicate Detection in turn relies on the existence of Sequence Identifiers. Corrupted Data Detection can be realised using a cyclic redundancy check (CRC) or using Parity. Acknowledgements can be implemented using a Positive, Negative and/or Delayed scheme. Further a Poll-Status scheme can be used. The functions RTT Estimate and Timer Management are required by Acknowledgements.

Timer Management can be realised using the Setting, Clearing, Expiry, Clear & Set and/or the Expire & Set functions. The Round Trip Time (RTT) Estimate function depends on Sequence Identifiers. This in turn can be implemented using Time Stamps or Sequence Numbers.

To support Expedited Data the functions Buffer Management and Error Control are required.

Zitterbart [93] has also developed a protocol dependency graph. However it has been limited to only two levels, atomic functions and functions that use these to directly support the service criteria.

3.1.1 Combining the Protocol Functions

A protocol is then synthesised by combining the protocol mechanisms of figure 3.1. When combining functions, the resulting protocol needs to provide sufficient func-

tionality for the application. If this is not the case, then the missing functionality will need to be implemented at a higher layer of the communication model, with possibly a greater performance cost than at the lower layer [88].

Due to the hierarchy of figure 3.1 it may be possible to support some protocol functions without a processing penalty. For example, if Ordered Delivery is required, then Buffer Management and Sequencing must be provided. Thus it is possible to provide Segmentation & Reassembly for free as its introduction does not require any other extra protocol functions to be enabled.

The choice of algorithms to implement each protocol function will be based on ILP considerations, end system constraints, and the service provided by the underlying network. Firstly, as ILP will group Protocol Functions together, it will be possible to support some of the protocol mechanisms with minimal overheads. Therefore the inter dependencies of the protocol mechanisms need to be established.

When creating a protocol the consequences of the two effects above must be considered so that an efficient protocol with the maximal functionality subject to the application's requirements can be constructed.

3.1.2 Mapping of QoS Requirements into Protocol Functions

The application's requirements need to be specified in terms of high level uses, for example email or low resolution video. These requirements are then mapped into a set of protocol functions. In order to perform this mapping, the generic set of QoS requirements developed by the OSI95 project [24] can be used as a starting point.

These QoS requests will be associated with either real time or non-real time

applications. Using this knowledge, the synthesised protocol will be determined from the Class of Service. Once the service class is determined then the required Protocol Mechanisms will be chosen. A possible mapping, similar to the one reported in [93] has a number of classes - Real Time Unreliable (RT-U); Real Time Reliable (RT-R); Non-Real Time Unreliable (NRT-U) and Non-Real Time Reliable (NRT-R) - as shown in Table 3.2.

Mechanism	RT-U	RT-R	NRT-U	NRT-R
Segmentation and Re-assembly	O	O	O	O
Ordered Delivery	M	M	O	M
Loss Management	O	M	O	M
Congestion Management	O	M	O	M
Data Flow Management	O	M	O	M

Table 3.2: *The Requirements for the Various Protocol Functions.*
Key:
O - Optional.
M - Mandatory.

Work by Diaz [27] has developed QoS parameters based on a three-tuple which consists of ordering, reliability and timing, as illustrated by figure 1.9 can also be used for this purpose. The three-tuple can be mapped into the protocol functions of figure 3.1 in the following way:-

ordering is performed by the *sequencing* protocol function,

reliability is performed by *loss management*, and

timing is performed by *rate modulation*.

3.1.3 Discussion

This section has decomposed end-to-end protocols into its component protocol functions. Guidelines for combining these functions to synthesis application specific

functions have also been presented.

However, the effect on the performance of a factorised protocol has not been ascertained. Infact, it is possible that a factorised protocol may have a worse performance than an integrated one. Thus performance experiments are required to ascertain its effects.

3.2 The Testbed to Evaluate Protocol Performance

Investigations into the overheads of end-to-end protocols and its component functions were performed by observing the operation of an implementation based on the TCP/IP protocol suite.

The experiments were performed using Mach, a micro-kernel based operating system [7]. Two hardware platforms were used; a 486DX-33MHz (also referred to as i486) and a Dec 5000/240 (also referred to as mips). These platforms were chosen to evaluate the effect of hardware evolution on protocol performance – the 486 is a CISC processor with a SPECint[1] rating of 12.8 while the Dec is a RISC processor with a SPECint rating of 27.9 [26].

Unless otherwise stated, all the experiments involve one way data flow (from a client to a server) using a dedicated Ethernet, and a transmitter and receiver buffer size of 60 Kbytes each. The side effects of this experimental setup are twofold. Firstly, the probability of collisions on the Ethernet is minimal. Secondly, there is no opportunity for the protocol to benefit from piggy-backing of acknowledgements.

[1]SPECint is a measure of the processor's speed with integer operations.

3.2.1 The Protocol Server

It is not practical to implement and experiment with a factorised protocol fully within the kernel of a monolithic operating systems (such as UNIX) due to the time required to compile, link and reboot the system each time changes are performed.

A better approach is to either implement the protocol in user space using a protocol function library [57, 84] with macro-kernel or micro-kernel operating system support. Alternatively the protocol can be implemented as a server within a micro-kernel based system. The second approach was chosen because of the accessibility of the software and operating system support at the time that this project commenced[2].

The standard distribution of the Mach operating system comes with the protocols built into the UNIX server. For these experiments, the code for the protocols was removed from the UNIX server and a stand alone protocol server was developed. This frees the operation of the protocol server from the framework of a file system, process manager, etc. However, the protocol server can also take advantage of the UNIX server by using its file system support when necessary. In addition, the development of the software is also unrestricted, as the protocol server is just like any other application that can be terminated and restarted without rebooting the computer. Further, it is virtually hardware independent, as the only code that is machine dependent is an optimised version of the checksum calculation.

The organisation of the protocol server, in the context of the overall system, is illustrated in figure 3.2. The server operates on top of the Mach kernel, at the same level as the UNIX server, in user space.

[2] About halfway during the experiments, a version of Mach was released that supported user level protocol implementations - known as the proxy option. Except for connection establishment and closing, the protocol operations were implemented by a library that is executed by the application [57].

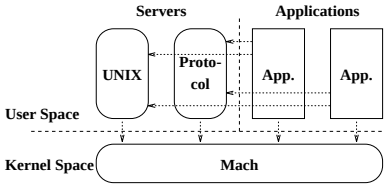


Figure 3.2: *The Interrelationship between Mach, the UNIX Server, the Protocol Server and an Application.*

The protocol server provides the full functionality of the TCP/UDP/IP protocol suite. This includes the majority of the BSD socket interface, and low level functionality of the protocol, such as the address resolution protocol (ARP) [80].

The protocol server was created from Mach's UNIX server by using the TCP/IP software (directories `server/netinet` and `server/net`). This software required some of the support functions provided within UNIX [52], such as mbuf, timer management and synchronisation (from the `server/sys` directory). Further, the IPC interface that was involved with the protocol's operations was re-used (directories `server/uxkern` on the server side and `emulator` on the client side). The directories `server/i386` and `server/mips` (for the respective machines) provided optimised code for the checksum calculation. Finally, the software needed to be initialised properly.

Thus it can be seen that the code for the protocol server was derived from the source code of Mach's UNIX server. This in turn was derived from the BSD code. Any change in performance for any of these implementations is due to the differing infrastructure that supports the implementation.

3.2.2 BSD Socket Interface

A consequence of the experimental approach (of only comparing the different operating system environments by using existing code) is that the protocol server has virtually a BSD socket interface. This was not a requirement of the project, but it has proven to be useful due to the availability of a very large range of applications that use this interface.

The main difference between the BSD socket interface, and the experimental system's interface stems from the protocol server's inability to recognise processes. Thus the protocol server behaves in a different manner to the *fork()* command [48] than expected by programs using the BSD socket interface[3].

Apart from the minor modification to the application's actions after a *fork()*, all that is required to use the protocol server is that the code is compiled, and linked with the correct library.

3.2.3 Instrumentation

In order to do performance experiments with the protocol server, software probes were used to instrument it. These probes are similar to those used by Papadopoulos

[3]The *fork()* command in UNIX creates a new process. Part of this creation is the duplication of file descriptors. (A file descriptor is a pointer into the kernel that eventually translates into an open file, or socket). Whenever the file descriptors are duplicated, the kernel is aware of it, so to close a socket, both processes after the fork() command must close their file descriptors. However, with the protocol server, the file descriptors are in a global array (because the protocol server is unaware of processes), thus after a fork(), the file descriptors are not duplicated. Thus either process can (and will) close the socket alone. This is not the same as a true socket interface. The problem is not major, and can be fixed in at least one of two ways. Firstly the fork() system call can be re-written to communicate to the protocol server. This way, the protocol server will be aware of processes. Otherwise, as performed in this project, the application will need to be modified so that only one of the two processes (ie the parent or child) uses, and closes the socket. The other process just ignores it.

[66] which consist of a few lines of code that allow critical events to be *time-stamped* and logged onto RAM, using a circular linked list. Further, variables such as the size of the packet being dealt with, or the length of the queue associated with the probe are also saved. At the end of an experiment, the log is copied to disc.

There are a few differences in the implementation of a probe used in the protocol server compared to those used by Papadopoulos [66]. Firstly, an array of probes was used instead of a circular linked list. Secondly, all packets in the protocol server were logged (rather than only the ones associated with a new socket option (`LOGGING`) set). This is because the protocol server was only used to perform experiments, and so it needed to log all packets.

Probes were also placed in the Mach kernel. These probes are identical to those described in [66].

The overhead of a probe is minimal. All that is involved is the addition of a timestamp and copying the monitored variables into an array, which is then copied to disc at the end of the experiment. The 156μs resolution *time-stamp* was obtained using a mapped timer, rather than a system call, which only requires a variable look up to be performed. Figure 3.3 shows the macro used to place a probe into the software.

The system clock was modified to give this timestamp at a resolution of 156μs by increasing the rate of interrupts. The interrupt service routine was modified so that the clock would be updated every interrupt, but all other interrupt functions were performed at the original rate, thus ensuring no significant extra overhead.

```
#define new_Probe_Point(theID,thePacketSize,theQueueLength)
{
  struct ProbeInfo* ptr;

  mutex_lock(myProbeLock);

  if (nextProbe ≥ endProbe)
  {
    mutex_unlock(myProbeLock);
  } else {
    ptr = nextProbe++;

    mutex_unlock(myProbeLock);

    FastReadTime(ptr→timeSec, ptr→timeUSec);
    ptr→id = (theID);
    ptr→packetSize = (thePacketSize);
    ptr→queueLength = (theQueueLength);
  }
}
```

Figure 3.3: *The Macro that was Used to Place a Probe in the Software. The slash at the end of each line has not been shown for clarity.*

3.2.4 Operation of the Protocol Server

The queueing model for the operation of TCP within the SUN OS used by Papadopoulos [66] can be used to describe the operation of the protocol server. The model assumes that data is either being processed, or it is queued to be processed.

The main differences between the model used in this thesis and that of Papadopoulos [66] stems from the different operating system architectures. Within Mach, each IPC is modelled as a queue. Further, when data is removed from an IPC by a server or application, it is immediately queued again until it can be processed. Under normal operation (that is when not overloaded), the data is immediately removed from this second queue. The other difference of this model is that

the processing requirements of TCP and IP are separated.

Figure 3.4 shows the model used. Data generated by an application is passed to the protocol server in discrete units, using Mach's Inter-Process Communication (IPC) framework. Probe 1, as illustrated in the figure, has been placed in the application just before the IPC function is called. The data that is being transferred is queued by Mach, until the protocol server is activated. Probe 2 is located immediately after data is received by the protocol server. The data is then placed in another queue.

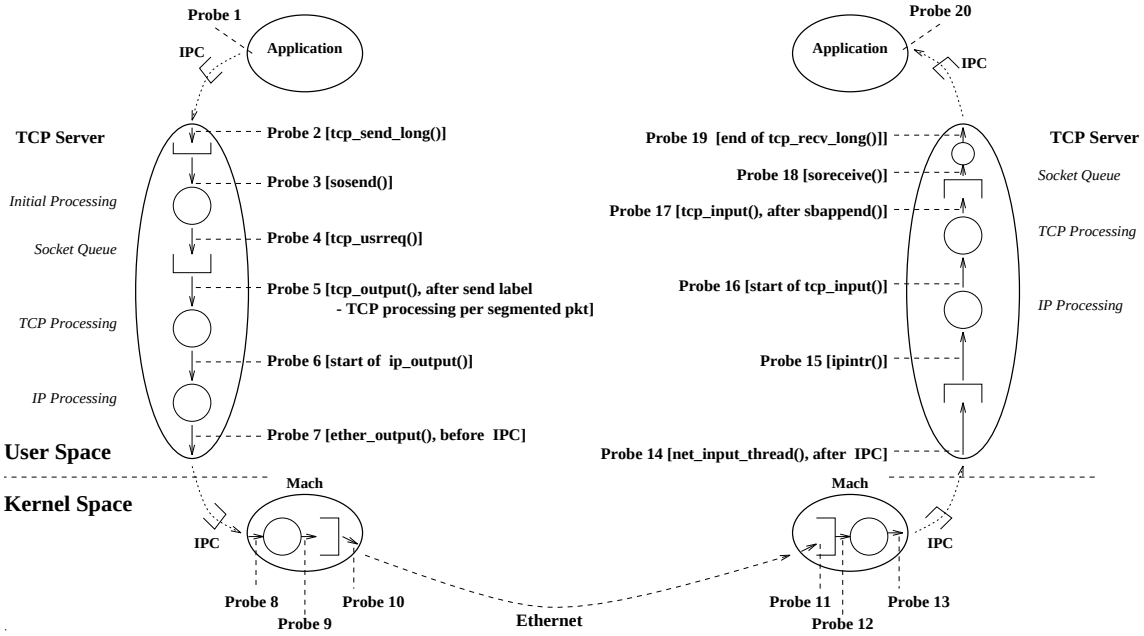


Figure 3.4: *The Model Used for TCP Operations.*

Probe 3 represents the start of processing within the protocol server. The application's data is copied into a socket buffer (using an mbuf [81]), and then the program executes the function `tcp-usrreq()`[4]. The start of transmission of an individual TCP packet is represented by probe 5. Probe 6 indicates the beginning of IP processing. Finally, probe 7 is used to monitor when protocol data is transferred

[4]The function `tcp-usrreq()` is the main command interface function into TCP.

to the Mach kernel.

The IPC into the Mach kernel represents another queue. After a packet is removed from the IPC (probe 8), the correct peripheral interface is determined. The packet is then placed on the peripheral's queue (probe 9). Finally, the start of the packet's transmission occurs at probe 10 [5]. If there is still untransmitted data remaining in the socket queue, the protocol server will continue execution - looping to probe 5.

Probe 11 represents the reception of a packet, which is queued, and then processed by Mach (probes 12 and 13). The start of the protocol server operation for incoming packets is monitored by probe 14. After further queueing, IP processing starts at probe 15.

Probe 16 monitors the transition from IP processing to TCP processing. After the TCP processing is completed, indicated by probe 17, the data is queued in a socket buffer. Probe 18 monitors the server side of this queue. The data is removed from the queue, and is prepared to be transferred the application at probe 19. Probe 20 indicates the reception of the data, via an IPC, by the application.

When one way data transfer experiments are performed then the probes activated in the reverse direction are a result of ACK packets only. For example, when probe 14 is reported by the transmitter, it implies that it has received an ACK packet from Mach. The ACK packets are created at probe 5 at the receiver, and will activate all probes up to 17 (inclusive) at the transmitter. No other probes are involved as ACKs do not directly interact with the application.

[5] Note, newer versions of Mach (on some architectures) have mapped the Ethernet device driver onto the user space. If this were done, the IPC queues and the processing from probes 8 to 13 (inclusive) would be eliminated [56].

3.3 TCP/IP Experiments

Experiments consisting of transmitting a single packet containing 1 byte or 1400 bytes of user data were conducted. The former enabled an evaluation of the protocol processing requirements without the significant overheads associated with data copying. The latter represents a full Ethernet packet. Finally, experiments involving the transmission of 1024 Kbytes of user data were performed. This allowed the full operation of TCP/IP to be observed.

The results of these experiments are described in the following subsections.

3.3.1 System Overheads

Primarily, the overhead associated with Mach arises from the use of IPC to transfer data between processes. There are two types of IPC associated with the protocol server.

One type passes data between the application and the protocol server. The IPC from probe 1 to 2 at the transmitter, and from probes 19 to 20 at the receiver are examples this type of IPC. A context switch from the application/server to Mach is required, followed by a *copy on write* buffer being set up, and finally a context switch from Mach to the server/application is performed. Mach may schedule other processes during this operation. Several projects are investigating methods of reducing the overhead associated with this IPC [31].

The second type of IPC is used to pass data between the protocol server and Mach; such as between probes 7 and 8, & 13 and 14. This operation requires another context switch, when the flow of control switches between the server and Mach. The buffer used in this IPC does not have the overheads associated with the copy on

write buffers. Newer versions of Mach allow the Ethernet device to be mapped into the protocol server's address space, which eliminates the overhead from probes 8 to 13 (inclusive).

The other system overhead is a result of the initial Processing in figure 3.4, which is required to convert a Mach IPC interface into the BSD UNIX interface that the code was originally designed for. Again if TCP was rewritten this overhead could be avoided.

Tables 3.3 and 3.4 quantify these overheads by looking at the transmission of only one Ethernet packet. This packet has either 1 byte of user data, or 1400 bytes of user data to represent a full packet.

Appendix A analyses the error introduced due the coarse grain timer used, and concludes that it is equivalent to a normal random variable with a standard deviation no greater than 3.9μs. The calculated standard deviations confirm the analysis in the appendix.

The protocol overheads are as expected. However some anomalies were observed.

The TCP to Mach delay (probes 7 to 8) is much less than that of Mach to TCP (probes 13 to 14). This is because the IPC into Mach (probes 7 to 8) is implemented as a Mach system call. The other delay is a true IPC operation.

The single IPC from Mach to the protocol server (probes 13 to 14) is less than half the delay between the user and the protocol server (probes 1 to 2 and 19 to 20) which both represent 2 IPC operations (one IPC to Mach, and then one out of Mach). However, the IPC between probes 1 & 2 and 19 & 20 requires more processing because they also involve a context switch into Mach, setting up copy on write buffers, making scheduling decisions, and then resumption of the new process.

The time taken between probes 7 and 8 (TCP to Mach's transmission queue) is

		Between Probes	1 Byte Delay (μs)	1400 Bytes Delay (μs)	Return ACKs Delay (μs)
System	User to TCP	1 & 2	897.39	859.17	
	Init Queue (Tx)	2 & 3	85.80	101.79	
	Init Proc (Tx)*	3 & 4	358.41	515.97	
	Socket Queue (Tx)	4 & 5	38.22	28.47	
	TCP (Tx)*	5 & 6	151.71	384.54	39.78
	IP (Tx)	6 & 7	153.66	168.87	157.56
System	TCP to Mach	7 & 8	83.46	296.01	125.58
	Mach Proc (Tx)	8 & 9	19.11	11.31	1.56
	Mach Queue (Tx)	9 & 10	42.90	63.96	14.43
System	Mach Queue (Rx)	11 & 12	97.50	125.58	94.38
	Mach Proc (Rx)	12 & 13	122.46	127.14	125.58
	Mach to TCP	13 & 14	193.83	341.25	193.83
	Init Queue (Rx)	14 & 15	167.31	117.00	155.61
	IP (Rx)	15 & 16	74.10	64.35	85.80
	TCP (Rx)*	16 & 17	382.59	588.51	208.26
	Socket Queue (Rx)	17 & 18	363.48	381.03	
System	Socket Proc (Rx)*	18 & 19	774.93	1116.96	
	TCP to User	19 & 20	683.67	652.08	

Table 3.3: *i486 Platform Processing and Queueing Delays.*
The delays have been averaged from 400 packet transmissions. The socket option `TCP-NODELAY` *was set. The 1 byte and 1400 byte cases are for the data path only, and do not include the acknowledgements. The ACK case is for the return acknowledgements.*
** These entries include a data copy or checksum calculation.*

larger for the ACK packets on the i486 than the 1 byte user data packets (which differ in size by only 1 byte). However, the processing within Mach (probes 8 to 9) is significantly less for the ACK packets. For the mips machine, there is no significant difference between ACK packets and the 1 byte user data packets.

An explanation for this could not be found. However it is suspected that it could be related to the priority of the threads performing the operations. The timer thread (which generates the ACKs) has a priority of four, where as the thread that transmits data runs at a priority level of five (lower priority).

		Between Probes	1 Byte Delay (μs)	1400 Bytes Delay (μs)	Return ACKs Delay (μs)
System	User to TCP	1 & 2	741.76	755.18	
	Init Queue (Tx)	2 & 3	87.23	88.45	
	Init Proc (Tx)*	3 & 4	297.68	528.87	
	Socket Queue (Tx)	4 & 5	12.51	8.24	
	TCP (Tx)*	5 & 6	89.67	249.49	41.18
	IP (Tx)	6 & 7	97.30	102.18	107.67
System	TCP to Mach	7 & 8	89.06	356.85	94.25
	Mach Proc (Tx)	8 & 9	12.51	13.42	15.56
	Mach Queue (Tx)	9 & 10	12.81	16.47	10.98
System	Mach Queue (Rx)	11 & 12	67.71	78.39	68.63
	Mach Proc (Rx)	12 & 13	74.12	78.39	86.31
	Mach to TCP	13 & 14	122.31	191.85	119.26
	Init Queue (Rx)	14 & 15	126.58	96.69	154.94
	IP (Rx)	15 & 16	52.16	42.40	57.65
	TCP (Rx)*	16 & 17	282.74	434.93	173.55
	Socket Queue (Rx)	17 & 18	224.18	225.70	
System	Socket Proc (Rx)*	18 & 19	418.77	611.22	
	TCP to User	19 & 20	401.99	351.67	

Table 3.4: *Dec 5000/240 Platform Processing and Queueing Delays. The delays have been averaged from 400 packet transmissions. The socket option* `TCP-NODELAY` *was set. The 1 byte and 1400 byte cases are for the data path only, and do not include the acknowledgements. The ACK case is for the return acknowledgements.*
** These entries include a data copy or checksum calculation.*

When changing the processor platform from a CISC to a RISC processor of about twice the SPECint rating, the system delays are virtually unchanged at the transmitting side of a connection (probes 2 to 4, and 7 to 10). However, the Dec 5000/240 has a lower system overhead than the i486 when receiving data and transferring it to/from the application (probes 1 to 2, 11 to 14 and 18 to 20). This observation is confirmed by Bershad [6] who points out that the performance of a Mach IPC is dominated by the speed of the processor.

3.3.2 Protocol Overheads

The protocol overheads are due to queueing and protocol processing. To quantify these overheads experiments involving the transmission of only one Ethernet packet with 1 byte of user data were performed.

As expected, these results show that for one byte transfers, the protocol processing on the i486 is consistently greater than that of the Dec 5000/240.

The TCP processing overhead of the ACK packets is significantly less than that of the 1 byte user data packets both at the transmitter and receiver[6]. This is because of the overheads associated with data transfer protocol functions. The receiver explicitly tests if a packet is a pure ACK packet, and if so, then fully bypasses the code associated with receiving user data.

A comparison with the results from Papadopoulos [66] is given in table 3.5. The different experimental test-beds have approximately the same processor speeds, and differ in only two respects. Firstly the 486 is a CISC machine, while the SPARC is a RISC. Secondly, Mach is a micro-kernel while UNIX is a monolithic kernel[7].

When comparing the extrapolated values with that of Papadopoulos it can be seen that the sending side takes about the same time to process TCP/IP in both cases. However, firstly the receiving side takes much longer to process TCP/IP within the protocol server. Secondly, the queue delays using Mach are far greater.

When looking at both the system and TCP/IP processing together (under Mach) it can be seen that the protocol processing is less for the faster processor, but only some aspects of the system's overheads have reduced. Further research is need to

[6] The Mach implementation of TCP always sets the TH-ACK flag when TCP is in the established state. Thus all the TCP messages timed have a valid acknowledgement field.

[7] Despite the SPARC and mips machines both being RISC based, a comparison has not been made between them because processors run at different speeds.

investigate the trends.

	Protocol Server			Papadopoulos
	(1400 Bytes/packet)		Extrapolated	(1024 Bytes/packet)
	Between Probes	Delay (μs)	Delay (μs)	Delay (μs)
User to TCP IPC*	2 to 5	646	602	280
Protocol (Tx)	5 to 7	553	487	443
Protocol to Ethernet IPC	7 to 8	296	239	40
Mach to TCP IPC	13 to 15	458	432	113
Protocol (Rx)	15 to 17	653	600	253
TCP to User IPC*	17 to 19	1498	1401	412

The Papadopoulos column taken from [66].

Table 3.5: *A Comparison of the Protocol Server i486 Results with Papadopoulos. The protocol server is running on an i486, while Papadopoulos' results are from a Sparc 1: these machines have approximately the same speed. However, the protocol server's packets are approximately 36% bigger than that used by Papadopoulos. The extrapolated values have been calculated to estimate the delay of a 1024 byte packet by taking the difference between the 1 byte delays and the 1400 bytes delays of table 3.3, multiplying this by $\frac{1024}{1400}$, and then adding the 1 byte delay.*
** These IPCs also include a data copy.*

3.3.3 Overall Operation

Bulk data transfer experiments were conducted to illustrate the protocol's overheads for applications such as file transfers. The experimental setup used is the same as before, and the results are summarised in table 3.6.

The results indicate lower throughputs than those reported elsewhere. For example, Papadopoulos [66] reports 7Mbps using a Sparc 1. The difference in throughput is believed to be due to two reasons. Firstly the extra two context switches required by a micro-kernel operation system reduces performance. Data travels from the interface card to Mach, then to the protocol server (one context switch) and finally to the application (two further context switches - one from the server to the Mach

Machine	Transmit Side Throughput Mbps	Receive Side Throughput Mbps
i486	4.965	5.448
mips	7.830	7.708
mips→i486	4.430	5.767

Table 3.6: *A Comparison of Bulk Data Throughputs. 1024 Kbytes of data were transmitted. The transmit side throughput is calculated from the first occurrence of probe 1 to the last occurrence probe 10. The receive side throughput is calculated from the first occurrence probe 11 to the last occurrence probe 20.*

kernel and one from Mach kernel to the application). A monolithic kernel requires data to travel from the interface card to the kernel and then to the application (one context switch). A similar path is followed in the reverse direction.

Secondly, it is believed that Mach has an ineffective utilisation of the memory cache, especially for artificial experiments where the user does not manipulate the data [64]. UNIX (a monolithic operating system) has a very high cache hit rate under these conditions. However, when the application uses the received data, UNIX's performance decreases significantly. This will not be the case with Mach, as the cache already needs to be refilled [64].

The reason that the transmitter and receiver throughputs vary slightly is due to the buffering within the system. Timing at the transmitter starts with the system call from the application, and includes socket, and Mach IPC buffering. Likewise the timer at the receiver is stopped when the last byte is received by the application. Again this involves both Mach IPC and socket buffering.

The bulk data throughput of the Dec 5000/240 is about 50% greater than that of the 486-33MHz machine. However, the SPECint rating of the Dec 5000/240 is about double that of the 486-33MHz.

The indications are that one of the reasons that a higher increase in throughput on the Dec was not achieved is because the computation speed of the processor is limited by the speed of the memory. This is supported by Pagels [64] who has shown, when using Mach, that only 11% of mbufs are fully within the cache during the Unix server's processing of UDP packets. That is, the cache is of very little benefit.

Table 3.6 also provides figures for connecting different machine types together (mips→i486), in which a faster server is connected to a slower client. This illustrates the effects of system parameters on the operation of the communication subsystem.

A minor modification was required to slow start algorithm for this configuration. Normally the initial congestion window is set to the size of the send socket buffer (60 Kbytes in this case), while the threshold is set to 65535 bytes[8] (in the function `tcp-newtcpcb`). These values were both changed to be 23360 bytes (corresponding to 16 maximum sized TCP segments, and also the maximum length of a Mach IPC queue).

The reason that these changes were made is because the original values would cause the IPC queue from probe 13 to 14 to overflow at the i486 (the slower machine). This would then cause the slow start threshold to be set to the size of 5 TCP maximum segments. Over a long time, the threshold would increase (in one segment increments) until it reached the size of 16 maximum TCP segments. However the throughput of the transmitter is unnecessarily throttled until the threshold reaches 16 segments.

Figure 3.5 gives a snap shot of the operation of the protocol over time while transmitting 1024 Kbyte blocks of data.

[8] As the route from the sender to the receiver does not involve a gateway (the two machines are directly connected together), the initial congestion window and the threshold are not modified by

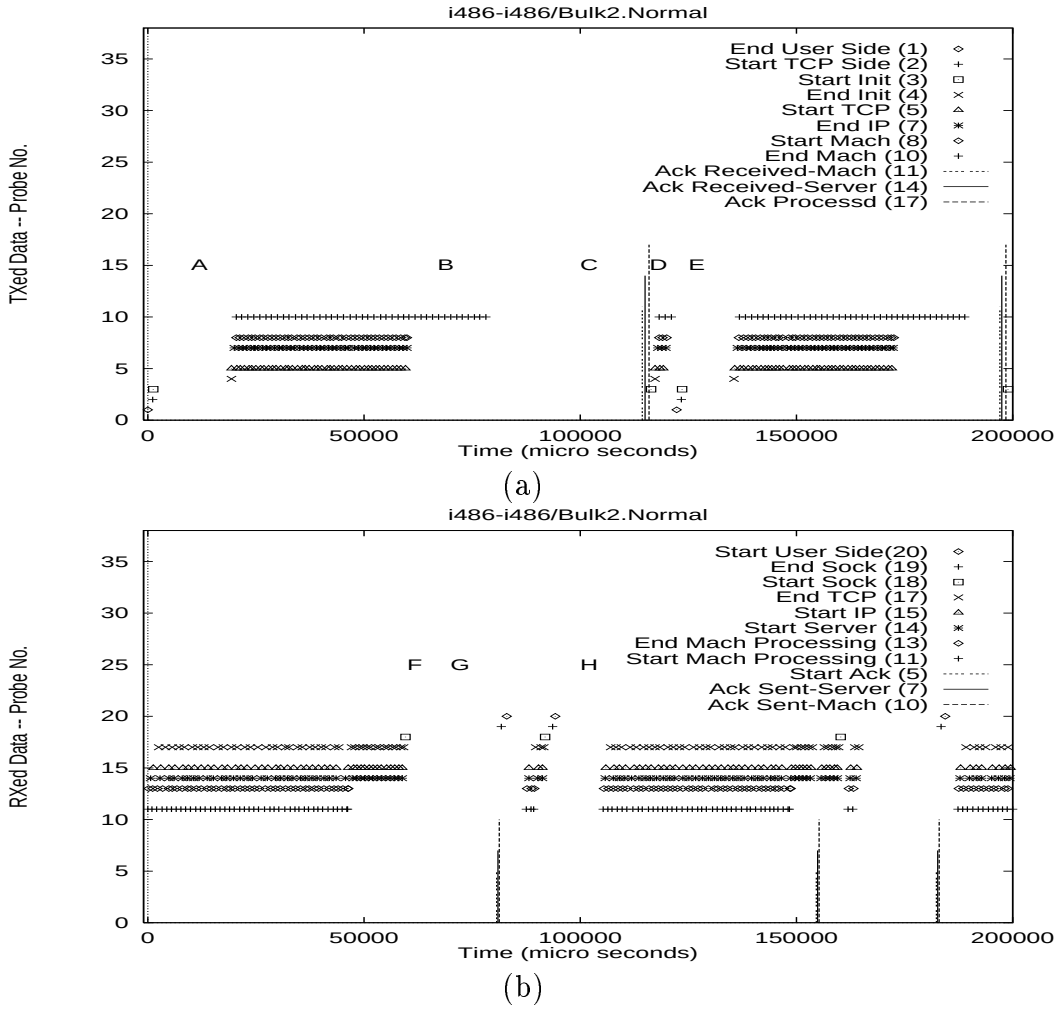


Figure 3.5: *The Flow Of Control of the (a) Transmitter Side and (b) Receiver Side of the Protocol Server.*
In the above experiments, the i486 processor was used and only the first 200000μs of a TCP connection is illustrated. For graph (a), the points represent events associated with transmitting data, whereas the lines represent returning acknowledgements. For graph (b), the points represent events associated with receiving data, whereas the lines represent the transmitted acknowledgements. (The absolute values on the time scale cannot be compared due to different clocks, but the differences can be). The values within brackets in the figures represents the probe number.
Key:
A. Probes 3 to 4: 64 Kbytes of data is copied into the mbufs.
B. Probes 4 to last 5 of of the group: the Mach Queue (Tx) is being emptied.
C. Probes 10 to 11: due to the effect of the window
D. The remaining 4K of data in the socket buffer is sent after an ACK is received.
E. A new 64K of data is copied from application.
F. Last probe 13 to last probe 14: the backlog of data is cleared from Mach.
G. Probes 18 to 19: Copy from mbuf to socket (60K).
H. Silence after probe 20: This corresponds with the mbuf copy (Key E).

Diagram (a) is for the transmitting side of the connection. The application generates data in blocks of 64 Kbytes, and passes it to the protocol server (probe 1). The protocol server receives the data (probe 2), and puts in into an initial queue. The data is later removed from this queue (probe 3) and then copied into the transmit socket queue (probe 4), using 4 kbyte cluster mbufs. This continues until the transmitter's socket buffer is full (60 Kbytes of data) or the IPC data is exhausted.

The protocol server starts the TCP processing by repeating a loop which contains probes 5 to 10 (inclusive). The maximum size of a TCP segment (packet) is 1460 bytes of user data (however, smaller segments will be sent if there is less than this amount of data in the socket queue). Probes 6 and 9 have not been illustrated because, on the scale of diagram, they appear to occur at the same time as probes 5 and 10 respectively.

This loop is interrupted by one of two events, either when the transmit socket queue becomes empty, or when an acknowledgement is received. In the first case, more data is transferred to the socket queue (probe 3; or if there is no more data to transfer, then more data is taken from the application - probe 1).

The other interruption to the loop between probes 5 and 10 is the reception of an ACK packet. (As these experiments involve only one way data transfers, data packets cannot be received by the transmitter). On figure 3.5 these events have been drawn as lines. Probes 11, 14 and 17 mark the arrival and processing of an acknowledgement.

Diagram (b) illustrates the receiving side of the connection. Data is received from the transmitter, passes through Mach (probes 11 to 13), and then via an IPC it is placed on an initial queue (probe 14). The packet is removed from this queue

---

the routing tables [81].

(probe 15), gets processed by TCP/IP and is then placed on the receive socket queue (probe 17).

At probe 18, data is removed from the socket queue and is passed to the application via an IPC (probes 19 then 20).

The receiver, which uses delayed acknowledgements, will only generate acknowledgements (probes 5, 7, & 10) when a percentage of the error control window is unacknowledged, or if there is no unacknowledged data and no packets have been received for a specified period of time. ACKs have again been represented as lines on figure 3.5 because the receiver will only transmit control packets.

It can be seen that flow control is slowing down the transmitter. This is because the receiver is processing the data at a slower rate than the transmitter. Another inefficiency at the transmitter is the processing delay between probes 2 and 3 (a data copy takes place). At the receiver, the delay between probes 18 and 19 is also due to a data copy.

When table 3.3 (the single packet experiments) are compared with figure 3.5 (continuous data flow experiments) a few points need to be remembered. Firstly the message size used from probes 1 and 4 in the table is 1400 bytes, as opposed to 64 Kbytes in the figure. Thus the delays between the probes which involve touching the data is larger for the figure. The TCP processing after probe 5 segments the data into messages of 1460 bytes, approximately the same size as used by the table. The average processing time from probes 5 to 6 to 7 and from probes 8 to 9 for continuous data flow are approximately the same as the values given in the table. The average queueing delays (probes 4 to 5, and 9 to 10) are greater for the continuous throughput experiments due to segmentation (probes 4 to 5), queues being filled up (for example region B in figure 3.5 results from the full queue between probes 9 and 10 being emptied) and the back pressure resulting from flow control

(for example region C in figure 3.5).

At the receiver the TCP/IP processing is similar for both the the single packet and continuous data experiments (probes 15 to 17). The Mach Processing delay (probes 12 to 13) is lower for the continuous data experiments. The reason for this could not be determined but it is speculated that it is related to cache effects combined with the thread priorities used.

As expected the queueing delays are longer for the continuous data experiments because the processor is not always immediately available to attend to packets in the queues – to case for the single packet experiments. Finally the queueing and processing delays from probes 17 to 20 are larger for the continuous data experiments because TCP will reassemble the data and will pass it to the application in larger sizes than that carried by a single IP packet.

An observation of figure 3.5 may imply that TCP/IP dominates the processing, while table 3.3 shows that this is not the case. It must be remembered that the table was calculated on a single packet being transmitted from application to application, where as the figure involves data being segmented from 64 Kbytes at the application layer to multiple 1460 byte (user data) packets at the network layer. It is a result of this segmentation that there are many probes resulting from TCP/IP processing in the figure.

The round trip time (RTT) measured from transmitting an IP packet (probe 7) to receiving its acknowledgement (probe 15 on the transmitting end), when delayed acknowledgements were disabled, was measured to be 7644μs on the i486 machines. Assuming that the Ethernet can transmit data at 10 Mbps, this delay is equivalent to 9555 bytes of data.

3.3.4 Mapped Ethernet Card

The version of Mach on the Dec machines allows the Ethernet card to be directly mapped into user space. When used, this effectively eliminates probes 8, 9, and 10 on the transmitter side and probes 11, 12, and 13 on the receiver side.

When experiments using a mapped Ethernet card were performed in the mips machines, the queueing and processing times between the other probes of figure 3.3 are essentially the same as table 3.4. Thus, as the processing at the receiver (which is the bottle-neck) has been reduced, this translates into an overall improvement in throughput. The transmitter achieved a throughput of 9.10 Mbps (compared to 7.83 Mbps without a mapped Ethernet card). The receiver has a throughput of 8.52 Mbps (compared to 7.71 Mbps).

3.3.5 Summary

The queueing model given by figure 3.4 was used to evaluate the performance of TCP/IP on two different platforms. The results agreed with literature [66] expect that the queue delays were greater for the results in this thesis. This is due to the operating system used, Mach a micro-kernel, as opposed to SunOS, a monolithic-kernel, which was used in [66].

When the speed of the processor was improved, the system delays were approximately unchanged, where as the IPC performance improved. This was as expected because the system delays are limited by the speed of a computer's memory where as the IPC performance is limited by the speed of the CPU [6].

The experiments involving bulk data transfers resulted in lower throughputs than reported elsewhere. This was due to the overheads introduced by Mach.

Increasing the speed of the processor did not result in a proportional improvement in the throughput of the system. This is because the speed of a computer's memory is not improving at the same rate as the CPU.

3.4 Factorisation of Protocol Functions

To determine the effects of various protocol mechanisms on system performance, the TCP code was modified so that the two most complex protocol mechanisms could be separated. This involved adding conditional statements to the TCP input and output functions so that the relevant segments of code were executed only if the functionality was required. The effect on throughput of the conditional statements is minimal.

In the following experiments, apart from TCP, TCP derivatives that had flow control, error control, and both flow control and error control factorised out (ie disabled) were used.

Tables 3.7 to 3.13 summarise the results, for bulk data transfers. In these investigations an additional set of experiments, which measured the performance between two dissimilar hardware platforms were also performed. This allowed two things to be observed; firstly the inter-operability of the protocol, and secondly the effects of flow control.

3.4.1 Error Control

Error control is very processor intensive for traditional TCP implementations, because it must be performed on every byte of data. This is highlighted by the fact that the UX server and protocol server both use a machine dependent optimised

Operation	i486 Mbps	mips Mbps	mips-i486 Mbps
TCP	4.965	7.830	4.430
No Error Control	5.481	8.912	4.805
No Flow Control	8.089	9.474*	–
No Error or Flow Control	8.311	9.411*	–

Table 3.7: *A Comparison of the Transmission Throughputs.*
1024 Kbytes of data was transmitted. Throughput is calculated from the first occurrence of probe 1 to the last occurrence probe 10.
** This value is limited by the speed of the network (Ethernet).*

Operation	i486 Mbps	mips Mbps	mips-i486 Mbps
TCP	5.448	7.708	5.775
No Error Control	6.190	8.820	6.389
No Flow Control	4.612*	6.940*	–
No Error or Flow Control	5.865*	9.223	–

Table 3.8: *A Comparison of the Receiver Throughputs.*
1024 Kbytes of data was transmitted. Throughput is calculated from the first occurrence probe 11 to the last occurrence probe 20.
** Data was lost.*

implementation of this function. Given the current trend towards RISC processors, and the rate of change of computer hardware (such as the CPU and memory) the processing overhead of error control is expected to increase over time [32, 82]. Table 3.9 shows the change in throughput when error control is removed by disabling the function call that calculates the checksum for each TCP packet that is transmitted (in `tcp-output()`) and received (in `tcp-input()`). Table 3.10 shows, as expected, that there is a negligible reduction of protocol processing (TCP/IP) for small packets without error control, but there is a significant improvement in protocol processing for large packets without error control.

For large packets, there is a better percentage improvement when error control is removed on the mips (RISC architecture) machine than on the i486 (CISC architecture) platform. However, the absolute improvement of the mips machine is 0.66

Machine	% Change in Throughput	
	Tx	Rx
i486	10.4	13.6
mips	13.8	14.4

Table 3.9: *The Effect of Error Control on the Throughput of TCP. The information for this table was derived from tables 3.7 and 3.8.*

Machine	User Data bytes	Tx Path			Rx Path		
		Error μs	NoError μs	Change % (μs)	Error μs	NoError μs	Change % (μs)
i486	1	305.4	301.5	-1 (-3.9)	456.7	449.3	-2 (-7.4)
i486	1400	553.4	311.2	-42 (-242.2)	653.9	448.5	-31 (-205.4)
mips	1	187.0	180.0	-4 (-7.0)	334.9	317.8	-5 (-17.08)
mips	1400	351.7	212.9	-39 (-138.8)	477.3	321.2	-33 (-156.2)

Table 3.10: *The Effect of Error Control on the Processing of TCP/IP. These results are averaged from 400 separate TCP segments. The Tx path is from probe 5 to probe 7. The Rx path is from probe 15 to probe 17.*

times that of the i486. The i486 has a SPECint rating about 0.5 times that of the mips machine. That is, according to the SPECint ratings, the mips machine has had a better improvement than expected.

This improvement can be attributed to the RISC architecture of the mips machine [82]. On these machines, the memory is much slower than the processor's speed (as measured by its SPECint rating). As discussed in §2.1.4, a high speed memory cache is used to provide data at a high speed. This approach is effective if the memory usage pattern is correlated. The removal of error control has resulted in an increase in the correlation of memory usage by removing the isolated references (within the TCP processing) to the data within the packets. The result of this is fewer cache misses because the data within the packet does not need to be loaded into the cache. Further, with error control enabled, some information initially within the cache would have to be re-loaded because it was removed to allow space for the data field of the TCP packet.

For both platforms, the absolute reduction in TCP/IP processing for the transmitter and receiver is the same. However, the percent improvement is greater for the transmitter because it performs less processing than the receiver.

A side effect of removing error control is the frequency at which acknowledgements were generated. For normal operation both the i486 and mips machines generated 21 and 23 ACKs respectively (measured by the number of activations of probe 14 at the transmitter). However, without error control, these values changed to 24 and 44 respectively. Thus the number of ACK packets generated by the i486 remained approximately unchanged, but the mips machine has nearly doubled the number of ACK packets generated.

A closer analysis of the mips results reveals that one data packet was lost during the transmission. The extra acknowledgements are announcing this lost data packet. If this loss had not occurred, then the throughput of the mips experiment without the checksum would be slightly higher. Despite this, the mips machine had a better overall (percentage) improvement by removing error control than the i486 machine.

3.4.2 Flow Control

Flow control is necessary in communication systems to ensure that the receiver is not flooded by the transmitter. Numerous schemes, varying in complexity, can be used to implement flow control. TCP uses a window based scheme. It is acknowledged that a communication system cannot operate without some form of flow or rate control. However to determine the effects of window based flow control on protocol performance, flow control was removed from TCP.

Flow control was disabled in the protocol server by ignoring the available window, and `tcp-output()` was modified to emulate an acknowledgement as soon as data is

transmitted. Computations, such as Van Jacobson's slow start algorithm [43] and timer operations were removed from `tcp-input()`.

Table 3.11 shows the change in throughput when the flow control mechanisms were disabled. The i486 transmitter experienced a dramatic improvement in throughout because it was no longer limited to the speed of the receiver. The mips transmitter's throughput also improved for the same reason, but the change was less due to the underlying network becoming a bottleneck. As expected, data was lost at the receiver for both platforms because they was not able to keep up with their respective transmitters.

Machine	% Change in Throughput	
	Tx	Rx
i486	62.9%	-23.1%*
mips 210.1%†	-10.0%*†	

Table 3.11: *The Effect of the Flow Control mechanisms on the Throughput of TCP. The information for this table was derived from tables 3.7 and 3.8.*
** Data was lost.*
† The Transmitter's throughput was limited by the speed of the physical network.

It is speculated that the drop in the receiver's throughput is due to the system's response to overload. This is because, with flow control, the transmitter needs to wait for the receiver to catch up (see figure 3.5). Without flow control, the transmitter now just sends packets faster than the receiver can digest them.

An analysis of the i486 results show that 736 packets arrive at the i486's Ethernet interface (probes 11 and 12), but only 175 undergo TCP/IP processing (probes 13 to 17). The rest *fall* off the end of the queue between probes 12 and 13 (Mach to the protocol server queue). This is after each packet has caused a hardware interrupt (from the Ethernet card), and been processed by Mach. The queue between probes 12 and 13 has a steady state length of 16 packets.

The reduction in processing of a TCP/IP packet is shown by table 3.12. The

trend is a small reduction in processing. However the changes are small relative to the quantisation error introduced by the timer which has been shown to have a standard deviation up to 3.9μs (see appendix A.

Machine	User Data bytes	Tx Path			Rx Path		
		Flow μs	NoFlow μs	Change %	Flow μs	NoFlow μs	Change %
i486	1	305.37	301.47	-1.28	456.69	461.76	1.11
i486	1400	553.41	538.59	-2.68	653.86	645.06	-1.35
mips	1	186.97	184.53	-1.31	334.89	332.39	-0.75
mips	1400	351.67	348.31	-0.96	477.33	467.87	-1.98

Table 3.12: *The Effect of Flow Control on the Processing of TCP/IP. These results are averaged from 400 separate TCP segments. The Tx path is from probe 5 to probe 7. The Rx path is from probe 15 to probe 17. Appendix A shows that these values have a standard deviation up to 3.9μs.*

A side effect of flow control is the generation, and subsequent processing of control packets. About 22 acknowledgement packets were generated for each of the experiments conducted with flow control enabled. Without flow control these packets are not generated and they do not contend for the Ethernet. Thus there is more time to process other aspects of the protocol's operation.

Thus the improvement in throughput when flow control is removed is due to the timing restrictions being relaxed at the transmitter (it no longer need to wait for the receiver) and because less packets (the acknowledgements) are generated. These could be said to be indirect reason because the time taken to process an individual packet has remained unchanged.

3.4.3 Error and Flow Control

Previous calculations have shown that, individually, removing error control and flow control improves the throughput of the protocol. Further, when both functions are

removed, the throughput is further improved at the transmitter. Table 3.13 shows that, at the i486 transmitter, the improvement is similar in magnitude to the sum of removing the individual functions from TCP. This is not the case for the mips transmitter because the throughput without flow control was limited by the available network bandwidth.

Functionality Functionality	% Change in Throughput			
	i486		mips	
	Tx	Rx	Tx	Rx
No Error Control	10.4%	13.6%	13.8%	14.4%
No Flow Control	62.9%	-23.1%*	20.1%†	-10.0%*†
No Checksum or Flow Control	67.4%	7.7%*	20.2%†	19.7%†

Table 3.13: *The Effect of both Error and Flow Control on the Throughput of TCP. The information for this table was derived from tables 3.7 and 3.8.*
** Data was lost.*
† *The Transmitter's throughput was limited by the speed of the physical network.*

This is not true at the receiver due to data being lost. For the i486 platform, the improvement in throughput without error and flow control is less than that of just no error control. The reason is similar to the case without just flow control – Mach's response to overload.

The processing of TCP/IP without either error control or flow control is shown in table 3.14. For all these experiments, the difference between the processing times of no error control and neither is minimal, when considering that the timer introduces a normal error of 3.9μs standard deviation, with a general trend of decreasing overhead. Likewise, for the one byte user data cases, the difference in processing overhead between no flow control and neither tends to decrease.

Machine	User Data bytes	Tx Path				Rx Path			
		TCP μs	NoError μs	NoFlow μs	Neither μs	TCP μs	NoError μs	NoFlow μs	Neither μs
i486	1	305.4	301.5	301.5	301.1	456.7	449.3	461.8	443.0
i486	1400	553.4	311.2	538.6	314.7	653.9	448.5	645.1	457.1
mips	1	187.0	180.0	184.5	168.1	334.9	317.8	332.4	322.1
mips	1400	351.7	212.9	348.3	201.0	477.3	321.2	467.9	311.4

Table 3.14: *The Effect of Error and Flow Control on the Processing of TCP/IP. These results are averaged from 400 separate TCP segments. The Tx path is from probe 5 to probe 7. The Rx path is from probe 15 to probe 17.*

3.4.4 A Comparison with UDP

Throughput experiments comparing TCP (without error and flow control) with UDP (without error control) were performed. To allow the results to be compared, the user message was broken into blocks of 1460 bytes in both cases. That is, the IPC between probes 1 & 2, and 19 & 20 carried a maximum of 1460 bytes per invocation. As a result the throughput for TCP is less than the values given by tables 3.7 and 3.8.

The results in table 3.15 show that a higher throughput is achievable from UDP both at the transmitter and receiver. This is as expected because UDP performs less functions than the TCP derivative used. Some of the extra operations performed by TCP include:-

- connection management,
- estimating the round trip time,
- maintaining a finite state machine,
- testing for urgent data,
- generating sequence numbers,

- testing for TCP options, and
- processing a larger header.

Protocol	Tx (Mbps)	Rx (Mbps)
TCP without error and flow control	3.32	2.33
UDP without error control	4.51	2.56*

Table 3.15: *A Comparison between TCP without error and flow control with UDP without error control on the i486.*
The TCP IPC between probes 1 & 2, and 19 & 20 carried a maximum of 1460 bytes per invocation.
** Data was lost.*

3.5 File Transfers

To put the results in context, a real application (a file transfer program) was modified to use the protocol server. The modifications required by the application software are given in §3.2.2.

A modified version of the Trivial File Transfer Protocol (TFTP) program [77] was used because of the ease of modification[9]. The modifications to TFTP were so that file transfers use TCP instead of UDP. This involved removing TFTP's error control and its support functions such as the retransmission timers. The packet formats used were modified by making the data packets the same size as the file being transferred. Finally, to permit the performance measurements, the overhead associated with disc access was removed by performing a memory to memory transfer.

[9] TFTP was used over FTP as the file transfer application, because it simplified the programming requirements. Both programs perform that same essential tasks, but FTP used library functions that converted its socket interface into a file interface. If FTP was to be used, these library functions would have to be rewritten.

The file transfer performs some presentation layer functions. The implementation supports two types of transfers, ASCII and binary. The ASCII transfers require the program to read each byte of data, and to modify it when an end of line is reached (both at the transmitter and receiver). Binary transfers do not require the data to be touched.

Throughput for these experiments was measured by the time between probes 101 and 102 on figure 3.6, divided by the file size. That is, throughput is measured only for the data exchange part of the file transfer program. Figures 3.7 and 3.8 show the results when file transfers were performed using TCP and various TCP derivatives. Further, files could be transferred either in ASCII or binary modes.

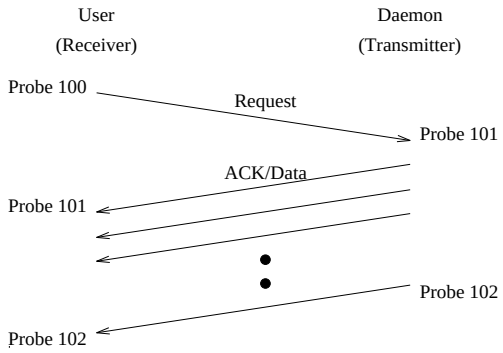
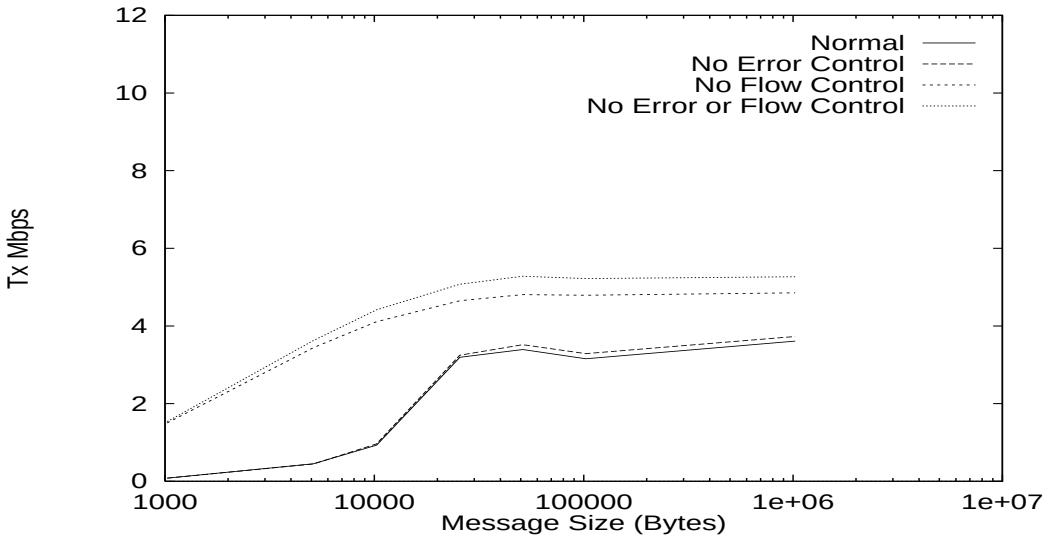


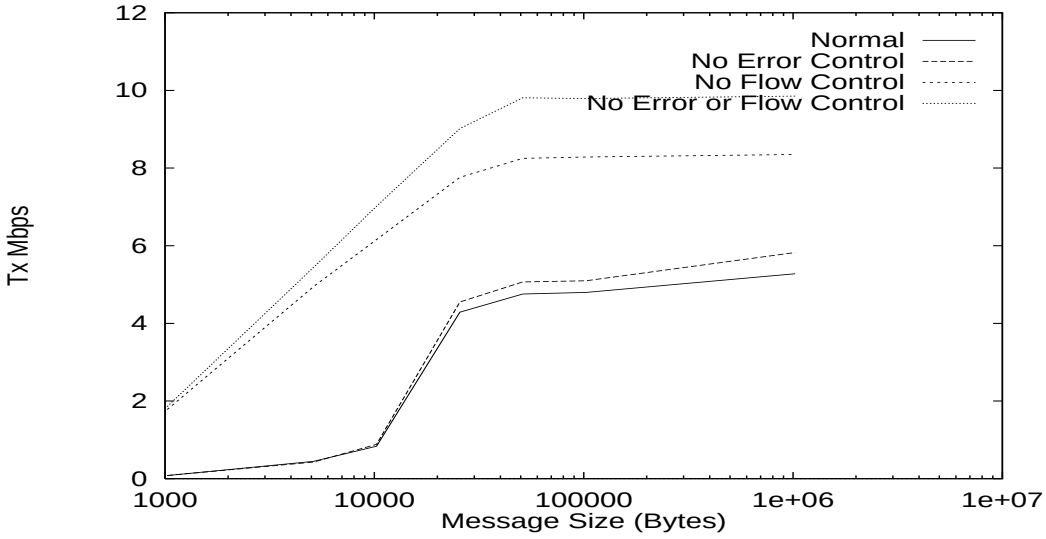
Figure 3.6: *The Operation of the File Transfer Program.*

The data being transferred represents typical text. Thus, when using ASCII mode, the percentage of carriage returns encountered in the file transfer is consistent with non-experimental systems.

The results show that the factorised protocols effect the overall performance of an application. The processing overhead of ASCII conversions is greater than that of error control. This is because the error control implementation has been optimised for the platform used, where as the ASCII conversion is generic for all platforms. Further, once a line feed has been encountered at either the transmitter or receiver, data is copied using byte operations. The error control implementation just reads

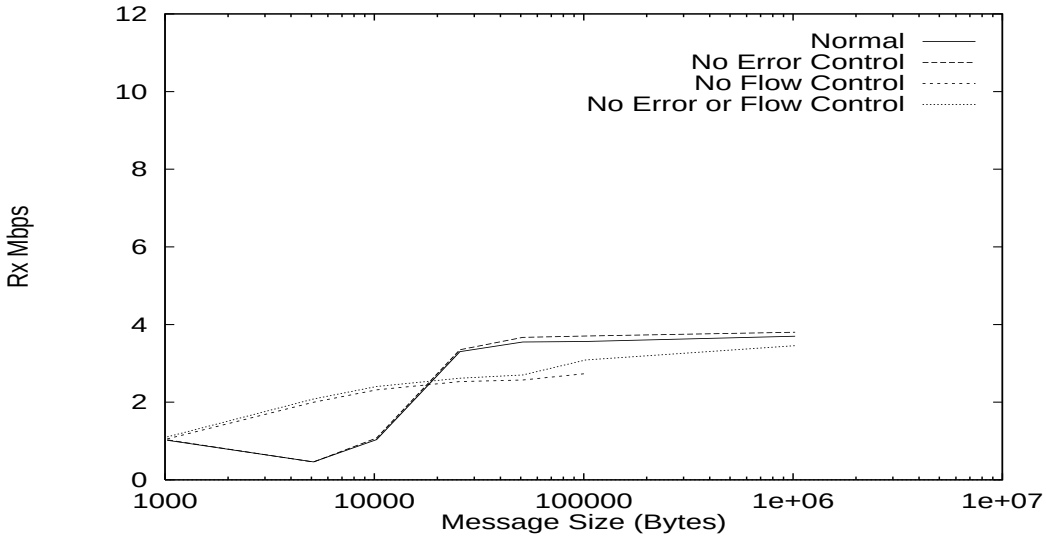


(a)

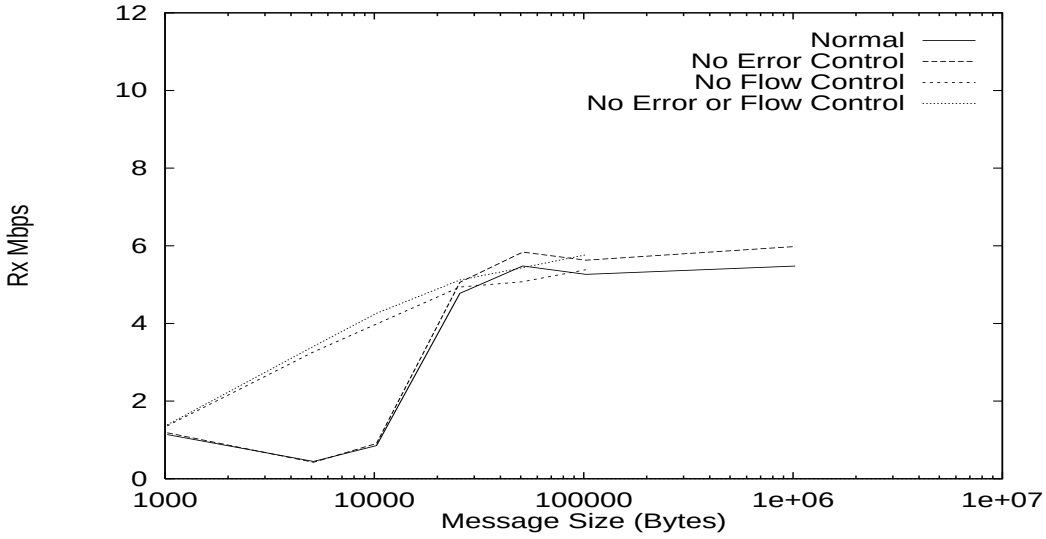


(b)

Figure 3.7: *The Throughput Achieved at the Transmitter for File Transfers Using the Different TCP Derivatives.*
Graph (a) is for ASCII transfers while graph (b) is for binary transfers.



(a)



(b)

Figure 3.8: *The Throughput Achieved at the Receiver for File Transfers Using the Different TCP Derivatives.*
Graph (a) is for ASCII transfers while graph (b) is for binary transfers on the i486 platform. The configurations without flow control have not been plotted for the 1000 Kbyte case because data was lost at the receiver.

the data, using the word size of the machine (16 bits for the 486 platform).

A local peak in throughput for both the transmitter and receiver with flow control is observable at approximately 60 Kbytes of user data. This corresponds to both the window size (60 Kbytes) used, and the maximum size of data (64 Kbytes) that could be transferred from the application to the protocol server. The throughput then improves for larger file sizes.

For large file sizes, the throughput of the binary file transfers corresponds with the values reported in Table 3.6.

3.6 Conclusion

A methodology for decomposing end-to-end protocols has been suggested. This decomposition was then used to modify the operation of TCP/IP to create TCP derivatives that had reduced functionality.

A queueing model of TCP/IP running under Mach was also developed. This model was used to evaluate the performance of TCP and the TCP derivatives. The results showed that when protocol functions are removed, the throughput of a protocol improves either directly, as is the case of error control, or indirectly, due to the reduction of packets being exchanged and timing restrictions being relaxed.

Further, these improvements were found to be independent. That is, when removing more than one protocol function, the resulting improvement in throughput is greater than removing the protocol functions independently.

It was also shown that these improvements in throughput were translated into better application level performance.

Finally, the queueing model was used to evaluate the two different processor platforms used. The results have shown that the RISC based processor is more sensitive to the data touching operations (such as error control) than the CISC based processor.

Chapter 4

Run Time Protocol Synthesis

The previous section has shown that the use of a factorised protocol is feasible. Whenever an application does not need certain functions, there is a performance benefit associated with a factorised protocol configured with only the required protocol functions.

This chapter will show that during the life of a connection, the protocol functions used may need to change. It will then discuss a method which allows run time protocol synthesis. The performance of such a protocol will then be evaluated to show that run time synthesis introduces no steady state performance penalty.

4.1 The Need for Adaptive Protocols

To quote La Porta, *"For a protocol to be successful in a high-speed environment, it must be flexible enough to supply several grades of service to the user over varying network topologies"* [49]. Four reasons for adaptivity have been identified in this

thesis; to support application diversity, to accommodate heterogeneity & flexibility, to improve QoS and to support changing application requirements.

4.1.1 Application Diversity

The diversity of network applications is growing. New applications are becoming feasible due to the cheap cost of computer power and the penetration of computer networks. As shown by table 4.1, the demands that each application makes on a network varies. Applications can either be real time (eg video playback) or non real time (eg file transfers). Real time applications may be of long duration (eg a video server) or short duration. Bursty traffic characteristics from real time applications must be supported within tight time constraints.

A video server requires a much higher bandwidth than an audio server. The network distortions permitted are also varied. Audio is not tolerant to jitter, where as video is more tolerant. Also the level of tolerable residue error, both from bit errors and lost packets, varies for each type of application.

Transport Service Class	Example Applications	Av. Thru-put	Burst Factor	Delay Sens	Jitter Sens	Order Sens	Loss Toler-ance	Priority Deliv-ery	Multi-cast
Interactive Isochronous	Voice Conversation	low	low	high	high	low	high	no	no
	Tele-Conferencing	mod	mod	high	high	low	mod	yes	yes
Distributional Isochronous	Video (comp)	high	high	high	mod	low	mod	yes	yes
	Video (raw)	V-high	low	high	high	low	mod	yes	yes
Real-Time Non-Isochronous	Manufacturing Control	mod	mod	high	var	high	low	yes	yes
Non-Real-Time Non-Isochronous	File Transfer	mod	low	low	N/D	high	none	no	no
	TELNET	V-low	high	high	low	high	none	yes	no
	On-Line Trans. Proc.	low	high	high	low	var	none	no	no
	Remote File Server	low	high	high	low	var	none	no	yes

From [74].

Table 4.1: *Application Transport Service Classes.*

Another difference with the applications is whether an interactive application is involved or not. An interactive network application allows processing optimisations such as piggy–backed acknowledgements to be used with the returning data.

Many other optimisations are possible for a given scenario. A general purpose protocol implementation that takes advantage of these scenarios would be complex. But, general purpose protocols are the only practical way to allow all equipment to be fully inter–operable over any network type.

An adaptive protocol will allow a custom protocol to be synthesised from a general purpose framework. This approach will support the inter–operable requirements of a network.

At present custom protocol must be designed for each application to obtain optimum performance, and it must be written by hand.

4.1.2 Accommodation of Heterogeneity and Flexibility

Another reason for an adaptive protocol is to maintain the quality of service (QoS) over variable levels of service, such as the network delay, jitter & throughput, or because of limited processing power or available memory. As stated by O'Malley "*This flexibility* [adaptivity] *is especially important when designing fault-tolerant systems because being forced to use a protocol that provides unnecessary or overly restrictive functionality can be extremely expensive*" [62].

An example of this case is a wireless network. The level of service provided by wireless links is determined from the conflicting requirements of bandwidth and power. The equipment used for the cellular network is power limited, meaning that there is a defined bit error rate [55]. Dynamic configuration can be used to adjust the protocols used so that "heavy weight" protocol functions can be used only when required. These heavy weight functions would require extra processing power, memory and/or bandwidth over the "lightweight" alternatives.

As the demands of a user changes, the resources consumed will vary. A flexible

protocol can reconfigure itself so that the best overall QoS can be provided to the applications with the limited resources available. An example of this situation is a video conference – such as IVS [86], and VAT [44]. If the parties in a two way video conference decide they need the expert opinion of a third person, then it would be natural to add another screen to the conference. Due to lack of space on the screen, the existing image size will need to be reduced thus allowing more image errors to be tolerated. Further, system limitation imply that one of the end computers cannot support two "heavyweight" video conferences. Thus it would be natural for the QoS of the existing conference to be down graded so that the third person can be added. A dynamically adaptive protocol allows this QoS change while maintaining the conference. When the conference returns back to only the original two participants, the QoS should also change back to its original settings.

4.1.3 Improving QoS through Flexibility

The QoS of an application can be improved with adaptivity. For example, parameters such as the observed end-to-end delay of a video connection can be minimised by having the application adapt to the prevailing delay of the network. Clark, Shenker & Zhang [20] have shown how such an adaptive application will produce improved performance over a static implementation using a conventional network. This is illustrated by a video conference. Adaptivity allows the end-to-end delay, from the transmitter's camera to the receiver's screen, to be minimised by varying the amount of buffering used. Sufficient buffering is provided to remove the effect of the network jitter. A non adaptive approach would need to operate at a higher end-to-end delay so that it can cater for the worst case jitter guarantee given by the network.

Looking at transport protocol mechanisms, [50] has shown that to optimise a

sliding window, the application must give "hints" to the protocol.

4.1.4 Supporting Changing Application Requirements

The QoS requirements requested by a user or an application evolve over time. This is best illustrated by two examples. Firstly, within a JPEG application, the initial data exchange (2 packets) must be ordered, however the following packets do not require order [28]. This ordering constraint is illustrated by figure 1.10. If an adaptive protocol is not used, then either unnecessary functionality is provided by the protocol (with its associated overheads), a low functionality protocol is used – with the application implementing the flexible protocol functions, or the application will use two protocols – TCP for the first two packets followed by UDP for the rest. These options result in poorly structured software because either the application implements part of the protocol, or more than protocol is required to transfer the data.

The second case involves the user deciding to change the QoS requested despite having sufficient resources available. For example, a user is watching a colour video conference, and he/she decides to save the extra (monetary) cost of colour by changing to black and white transmission. This results in a lower bandwidth. That is the decision to change the protocol is not based on resource limitations (which was the reason for change given in a previous example when a third person joined a video conference) but is solely determined by the users watching the conference.

4.1.5 Discussion

In order to satisfy these needs, it will be necessary to design new protocols for every new application. For example the error and flow control mechanisms may need to

be varied. Further the synchronisation scheme may need to be changed, and partial order will occasionally be needed.

A flexible protocol stack may be achieved by extending the existing protocols with options to dynamically select the required protocol functions, or by developing a generic framework that supports flexibility and an adaptive protocol realisation.

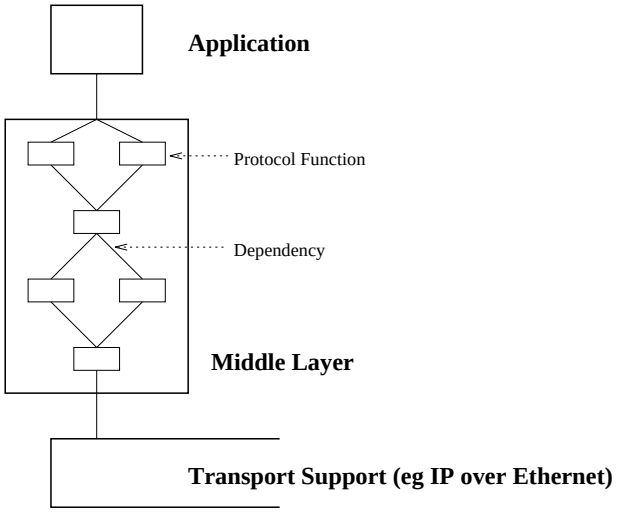
XTP [83] allows protocol selection, to a limited extent, by allowing error control to be selected on a per packet basis. Flow control functionality can also be disabled but not changed. However, all the possible protocol functions need to be compiled into the protocol. This leads to unnecessary code bloat and inefficiencies [41, 58]. An automated application specific protocol is unnecessarily restrictive, as only a limited predefined set of protocols can be available through the application's specification [25]. Thus, these two solutions will only enable an approximate matching of the application requirements.

In contrast, a dynamically adaptive protocol is able to maintain the data flow while it re-organises its support protocol functions. This approach takes advantage of the flexibility provided in the underlying network so that the (time) changing requirements of an application can be met in a continuous and efficient manner.

4.2 A Survey of Adaptive Tools

Adaptive protocols are based on factorised communication subsystems. The general model is composed of three layers. Applications exist at the top layer, while the network is at the bottom layer. The various protocol functions exist in the middle layer. This generic model is shown in figure 4.1.

Configuration of a protocol involves manipulating the dependency graph of the



Modified from [87].

Figure 4.1: *A Generic Model to Support Adaptivity.*

protocol functions in the middle layer. This allows the protocol to provide a very flexible service to the application. The dependency graph determines which protocol functions require the result of others.

This section will give details about three different adaptive protocols, namely Da CaPo, ADAPTIVE and F-CSS which attempt to provide means of achieving an efficient middle layer in figure 4.1.

4.2.1 Da CaPo

The Da CaPo (Dynamic Configuration of Protocols) calls the layers in figure 4.1 A, C and T respectively. A runtime environment to support the C layer has been implemented as a user level process within UNIX (SunOS). Issues that have been investigated in [87] include how to implement a protocol based on various protocol functions, and the importance of the order that the protocol functions should be performed. The solution proposed is that the environment should develop a partial

order dependency graph of the protocol functions. This is then used to determine the execution path of the environment.

To facilitate execution of the protocol functions, all protocol functions must have a unified interface. There are five aspects to this:-

- initialisation,
- data passing to/from the protocol function,
- control packets passed to/from the protocol function,
- static data,
- terminating the protocol function.

Another issue is the configuration process. For example, how does an application determine which protocol functions are required? This must be specified to the protocol environment. Thus an adaptive interface has been developed.

The configuration module within Da CaPo is known as CoRA. It operates by performing the following six steps:-

1. pre-decision (decide if a new protocol can be supported),
2. function elimination (exclude all functions that are not suitable, for example not including encryption if privacy was not specified),
3. T service selection (for example decide between Ethernet and ATM),
4. configuration (look for a configuration that complies with the application's request).
5. optimisation (try to improve the configuration), and

6. post-decision (decide if the new configuration will not adversely affect the existing ones).

To bound the configuration process, each protocol function is classified according to the following disjoint sets:-

- performance (throughput, delay and processing overheads),
- reliability (bit error/packet loss probability, etc), and
- miscellaneous (costs, security, etc. except performance and reliability).

For example, reliability is affected by protocol functions such as CRC, idle repeat request, selective retransmission and rate based flow control. Each protocol function also has attributes that quantify its resource overheads, and pre-requisite protocol functions.

The configuration process now requires a selection to be made from each set, rather than selecting from all the possible functions. As an illustration, if there are 4 possible protocol mechanisms for each of 4 different protocol functions (16 mechanisms in all), then there are only $4^4 = 256$ possible combinations (compared with $16! = 2.0933 \times 10^{13}$ overall combinations) that can realise the requested protocol.

4.2.2 ADAPTIVE

Another adaptive approach is fittingly known as ADAPTIVE (A Dynamically Assembled Protocol Transformation, Integration, and Validation Environment) [74]. It is a flexible protocol environment based on object oriented design concepts that correspond to the middle layer of figure 4.1. The project aims to solve four problems related to communication [74]:-

1. providing a high available throughput to the application,
2. support the functionality requirements of multimedia applications,
3. support a diverse set of underlying networks, and
4. provide a simple application interface.

However, the main focus of ADAPTIVE is to design and experiment with flexible and adaptive software architectures for the higher protocol layers.

To support this, the system has been designed to have five components within the middle layer, which are [75]:-

Stream: Performs initialisation and runtime operations.

Reactor: Performs de-multiplexing, temporal events (eg timers) and in/out events.

Service Configurator: Dynamically links in and out the services. Reconfiguration is permitted during the connection.

Concurrency: Responsible for management of the processes and threads.

IPC-SAP: Encapsulates the operating system, local interprocess communication and remote interprocess communication mechanisms.

Automated tools are also being developed to support this environment.

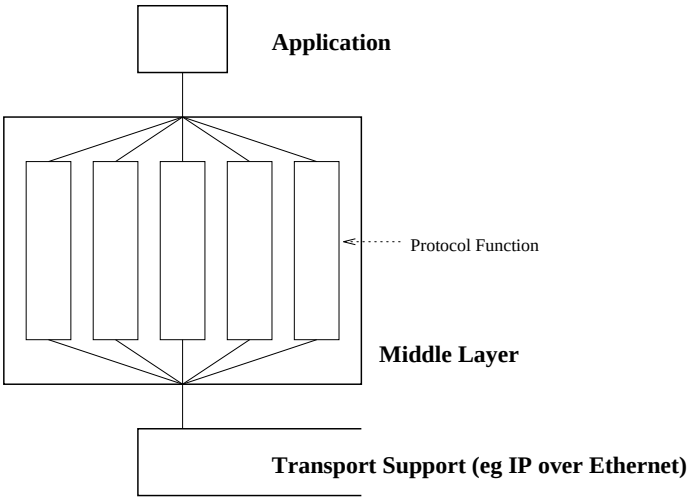
ADAPTIVE is configured by using pre-defined transport service classes.

4.2.3 F-CSS

F-CSS (Function - Based Communication Subsystem) is a flexible protocol design based on parallel implementation techniques [93]. Although designed a number of

years ago, it has never been implemented.

Again the model used is based on a three layer model, similar to figure 4.1. However, as shown by figure 4.2, within the middle layer the dependency graph of the protocol functions differs in that each protocol function can be considered (and implemented) in isolation.



Modified from [93].

Figure 4.2: *The F-CSS Model.*

The user interface to F-CSS is through pre-defined service classes. This has simplified both the system calls to the protocol and (more importantly) the configuration process because predefined configurations correspond to each service class. However, if none of the predefined service classes are suitable for a particular application then a custom protocol can be specified and configured.

To achieve this a QoS language has been developed which is known as the Function-Based Protocol Description Language (F-PDL). The language is used to specify the name, inputs, outputs and the operation (in C++) of the various protocol functions [93].

The protocol is configured in three steps:-

Validity: A check is made to ensure that the requested protocol is viable. For example the requested throughput is achievable.

Configuration: The protocol functions are configured and resources allocated. The information required for pre-defined configuration is stored in tables. Resources must be available at both ends of the connection. However for custom configurations, end-to-end negotiations are required to ensure the requested protocol functions are available (for example a particular encryption/decryption algorithm). This may be done either by a handshake, or implicitly.

Dependency Graph: A data dependency graph is generated to locate the synchronisation points of the protocol functions.

4.2.4 Protocols Tailored to an Application

An alternate method to customise a protocol is to implement a different protocol for each application, that is, to merge the top and middle layers of figure 4.1. Tailored protocols have some benefits over adaptive protocols in that they readily support ILP and ALF [21] concepts.

In addition, user level protocols [84] allow tailored protocols to be implemented efficiently. However, writing each protocol by hand will result in the inefficient use of computer programmers. This can be overcome if automated techniques can be developed.

One such approach is to allow the protocol to be specified in a high level language, and then compile and combine it with the application [1]. Resent work indicates that the performance of such a protocol is as good as hand-optimised application specific software [28].

Another approach to designing application specific protocols has been proposed by O'Malley [62]. The approach is to design a protocol graph for each application. The graph is composed of protocol functions, referred to as micro-protocols [62], and virtual protocols. A virtual protocol represents the control actions and decision points of the protocol graph. The protocol graph (and thus the protocol used) is configurable through the use of these virtual protocols. The configuration determined by the protocol graph, which is static. However, the paths through the graph can vary.

In either approach, the protocols are static in that unless specifically coded, the protocol will not respond to changes in the network environment or changes to the condition of the end host systems.

4.2.5 Discussion

As can be seen from the three adaptive models presented that each approach is aiming to develop a complete environment to support adaptivity. Further, each approach is not attempting to solve the complete problem. Da CaPo has concentrated on the configuration and execution path of a synthesised protocol. ADAPTIVE's model attempts to completely redesign the support environment for computer communications. F-CSS results in flexible protocols for parallel implementations. Further, this results in poor performance because it does not allow optimisations such as Integrated Layer Processing [21].

It has been shown by [18] that parallel implementation is not required to achieve high performance.

None of the approaches have investigated a modular method of allowing adaptivity that also permits efficient implementation. Further, the concept of an application

specific protocol is not supported.

Application specific protocols allow a tailored protocol to be easily developed. However the current design of tailored protocols do not support adaptive protocols [25].

4.3 Developing Runtime Synthesis

There are numerous ways of implementing run-time synthesis from a given protocol function library, such as:-

Embedded Conditional Statements: Conditional statements are placed within the body of various functions. Parameters used by these statements need to be stored as global variables, state variables for each connection, or are passed as parameters to the functions. This approach has been used in the implementation of XTP.

Static Linking: This involves using function pointers to access the desired protocol functions which are linked into the executable during its compilation. If a different function is required, the pointer is changed so that it becomes associated with the new function. During normal operation, this approach does not introduce extra conditional statements. This approach is similar to the upcall pointer proposed by Clark [17].

Dynamic Linking: A similar approach to static linking except that the required protocol functions are also loaded into the executable program only when required. This results in reduced executable size. Further, it provides a method to expand the available set of protocol functions. The DLD library [89] is one method that permits this approach to be realised.

Any of these methods will produce efficient implementations. However, the use of embedded conditional statements leads to unstructured software because it does not require the function calls to be related to protocol functions. This limits further evolution of the protocol.

Static linking allows a better structured software by forcing each protocol mechanism to be implemented by a separate C function.

A dynamically linked approach also forces separate C functions for each protocol mechanism, but it also allows a large range of options to be available without a blow out in the size of the executable. It is for this reason that it was chosen. However, as one of the objectives of this thesis is to show that dynamically reconfigurable protocols are practical it would not have mattered which method was used.

This section will explain the operation, and associated overheads associated with self-modifying software, and in particular dynamic linking.

4.3.1 Runtime Code Generation

Runtime code generation involves a program changing its operation by modifying its executable image. Although it is not in common use today, there are no technical reasons preventing it [47]. In fact, runtime code generation was a common technique used in early computers. Reasons for its use was that it allowed better use of the limited amount of memory available, and it allowed programs to run faster.

As computers became faster, and memory cheaper, runtime code generation fell out of favour. Other reasons for its decline in use include:-

- a formal framework for its use was never developed,
- it was seen to hinder portability because high level languages did not support

its abstraction, and finally

- monolithic compilers encouraged a separation of compile time and run time.

During this process, it was forgotten that there is fundamentally no difference between the code segment and data segment of a program.

However, this technique has not been totally forgotten, and is still in use in some systems, for example incremental compilers, dynamic linkers and debuggers [47]. Further, modern computer architectures are replicating conditions similar to the early machines. Although memory is cheap and plentiful, high speed cache memory is limited and expensive [82].

The other reasons for the decline of runtime code generation have also been addressed. The use of templates has provided a formalism for this technique, while schemes exist that permit portability [47].

There are performance benefits in using runtime code generation because a compiler can generate better code when it has more information. All the information is available at runtime. For example, a variable may be a constant within a loop or function. This knowledge can be used to optimise the code through techniques like dead code elimination.

The use of templates is one method that allows these optimisations to be performed. A template is a machine dependent representation of some code containing *holes*. At runtime, customising a template involves copying it, and filling in the holes with *runtime constants*. This allows runtime constant propagation to be performed [53].

An example of this type of optimisation is illustrated. Suppose the expression $A^2 + AB + B^2$ is to be evaluated for many values of A while fixing $B = 2$ (because perhaps it is within a loop). Then it would be more efficient to evaluate $A^2 + 2A + 4$

[58].

To allow this optimisation to be supported a template is created for each case during compilation [53]. At run time, the executable decides which template to use, and then fills in the parameters (such as $B = 2$), and then causes it to be part of the inline execution of the program. However, to be efficient, the reduction in processing must be greater than that required to perform the optimisations.

An example of this approach is the Synthesis operating system [58]. Templates are associated with the device drivers and interrupt handling parts of the operating system. This approach has resulted in improved performance compared to traditional operating systems such as UNIX and is illustrated by noting that Synthesis has a fine resolution scheduler while maintaining good performance [58].

The use of templates allows fast runtime synthesis, but only permits a limited range of optimisations. Another method to allow runtime code generation is to modify the high level compilers to produce an intermediate representation of a program. This technique is known as deferred compilation. Ongoing work on this approach is showing positive results [53].

4.3.2 Dynamic Linking

Dynamic linking [90], a runtime code generation technique that uses templates, is different to the traditional approach to software development in that the various software modules are linked into the executable program (by itself) during run time, as shown by Figure 4.3. Traditionally, all the modules required in the executable are linked during compilation. The major advantage of dynamic linking is that a program can offer a very large range of options, but the executable remains small.

The Dynamic Link/unlink eDitor (DLD) package [89] was chosen because it not

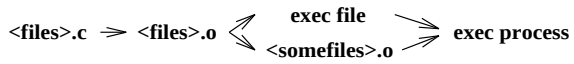


Figure 4.3: *The compilation process using dynamic linking. Not all of the* <SomeFiles>.o *need to be read into the executing process.*

only allows dynamic linking, but modules can also be unlinked when they are no longer required. Furthermore DLD is platform independent.

A similar concept has also been implemented in Lipto [33], an extension of the x-kernel.

4.3.3 The Cost of Dynamic Linking

There are two costs involved with dynamic linking[1]. Firstly, when a module is required, the object file must be read from disc, and then linked into the executing program. If this cost is unacceptable, a second process can be created to load the modules, allowing the first process to continually operate the protocol. Alternatively, if a certain function is continually being selected and then unselected, it can be kept in the computer's memory, and then only the linking operation needs to be performed. Section 4.5.1 will measure these costs.

Secondly, every function call to the dynamically linked part of the program involves a small overhead. This overhead is required for any adaptive software using dynamic or static linking.

If embedded conditional statements are used, then its evaluation also represents an additional overhead compared to a non-adaptive program. Further, parameters must be passed to the function to select an option. Figure 4.4 and table 4.2 show

[1] As shown in §4.4, a third cost exists. A negotiation phase occurs where both ends of a connection must confirm the availability of the protocol functions.

that the cost of passing an integer to a function is the same as the extra overhead associated with a function pointer. As the conditional statements also have an associated overhead, the conclusion must be that function pointers will produce more efficient software.

```
a = function1();
                    call  _function1__Fv,0  %call function
                    nop
                    st    %o0,[%fp+-0x4]    %save return value

b = function2(a);
                    ld    [%fp+-0x4],%o0    %pass parameter
                    call  _function2__Fi,1  %call function
                    nop
                    st    %o0,[%fp+-0x8]    %save return value

a = (*fn1)();

                    ld    [%fp+-0xc],%g1    %load address of function
                    call  %g1,0             %call function
                    nop
                    st    %o0,[%fp+-0x4]    %save return value

b = (*fn2)(a);

                    ld    [%fp+-0x10],%g1   %load address of function
                    ld    [%fp+-0x4],%o0    %pass parameter
                    call  %g1,1             %call function
                    nop
                    st    %o0,[%fp+-0x8]    %save return value
```

Figure 4.4: *A Comparison of the instructions involved when using indirect function calls.*
This code was created by using the AT&T C++ compiler (CC version 2.00.02) on a Sun SPARC-station. The variables `a` and `b` are of type `int`.

Function	C	Static/Dynamic Linking
`a = function1()`	3	4
`b = function2(a)`	4	5

Table 4.2: *A comparison of the costs of Providing Adaptivity. The costs have been calculated by looking at the number of machine code instructions generated by the AT&T C++ compiler when calling a function. The C column refers to calling the function without using pointers. The values for this table were obtained from figure 4.4.*

4.4 Other Overheads Associated with Adaptive Systems

Another overhead associated with adaptive software is the mechanisms required to ensure both ends of a connection can support the required protocol functions. For example if a certain encryption algorithm is requested by one end system, it must ensure that the other end can decrypt the data before the function can be used.

This requires a finite state machine to be added to both ends of the connection that negotiates the availability of the chosen protocol functions. Similar to connection setup (§2.3.4), two types of negotiations are possible, either a three way handshake or by using implicit techniques.

The three way handshake will result in a round trip delay between a request for a protocol function and its first possible use. The implicit techniques will result in received data being discarded whenever a requested protocol function is not supported by the receiver – a potential waste of end system and network resources. Either approach will also result in extra protocol processing.

4.5 Implementing Run Time Protocol Synthesis

To investigate the possibility of efficiently synthesising application specific protocols, the protocol server discussed in §3.2 was modified so that it allowed run time protocol synthesis. The term synthesis means that an end-to-end protocol can be made to work by combining its protocol functions (see §3.1).

To see the effectiveness of run time protocol synthesis, the instantaneous response of the TCP derivatives' throughput was investigated during variations of the flow control and error control capabilities within the life time of a connection.

It was assumed for these experiments that both end systems could realise the desired functions. That is, the overheads described in §4.4 do not apply to this system. This assumption is not feasible in a real system. A scheme that permits protocol selection without requiring all possible protocol functions available on all end systems will be described in the next chapter.

However, before these experiments were performed, the overheads of the dynamic linker were assessed.

4.5.1 Dynamic Linking

Experiments have been performed to ascertain the overhead of DLD, and are shown in table 4.3. The time delay required by the dynamic linker is large due to the disc access time. However, the table shows that the overhead of dynamic linking is dramatically reduced when the functions have already been loaded into the computer's memory[2]. The processing time is now insignificant when compared to the processing requirements of the protocol (table 3.3).

[2] From appendix A, it can be seen that the standard deviation of the results in table 4.3 is 3.9μs.

Function	From Disk		From RAM	
	Link Time (μs)	Unlink Time (μs)	Link Time (μs)	Unlink Time (μs)
Error Control On	28570	3205	57	49
Error Control Off	28749	3149	62	70
Flow Control On (Tx)	31698	3335	23	17
Flow Control Off (Tx)	31694	3400	30	27
Flow Control On (Rx)	32201	3327	30	34
Flow Control Off (Rx)	32118	3361	57	43

Table 4.3: *The Time Required to Perform Dynamic Linking on the i486 Platform. The measurements are averaged from 400 repetitions. When linking from RAM, the time to link is related to the order that the functions were initially linked from disc (the first one taking longer). This is an effect of the dynamic linker chosen. If a large number of functions are required, then a hash table could be used to remove this effect.*

The implication of requiring the protocol functions to be pre-loaded into memory is that, effectively, static linking is being performed. Thus, with this condition, there is no benefit of dynamic linking over static linking.

In order to allow dynamic linking from RAM, the DLD package needs a minor modification. Firstly the data structure `file-entry` was changed to include a new variable `active-ref-count` which counts the number of requests to link a given object file.

Secondly, the function `dld-file-link()`, whose purpose is to link in an object file, was renamed `dld-file-link-count()` and modified so that an object file is only loaded from disc if it is not already in RAM. Otherwise the variable `active-ref-count` associated with its instance is incremented.

Finally, the function `dld-unlink()`, whose purpose is to remove an object file from RAM was renamed `dld-unlink-count()` and modified so that it would remove the object file from RAM only if the variable `active-ref-count` indicates

that there are no other link requests for this object file. Otherwise the variable `active-ref-count` associated with its instance is decremented.

4.5.2 Modifications to the Protocol Server

The protocol server used in the previous chapter was modified to allow run time protocol synthesis. The modifications were performed so that the DLD (dynamic linking) package [89] could be incorporated into the server.

The results from table 4.3 imply that for efficiency, the protocol functions need to be pre-loaded into RAM during the protocol server's initialisation (effectively static linking).

For a non-experimental system, another thread can be created to perform the dynamic linking in parallel to the protocol's normal operation so that normal data flow is not interrupted. Further, by wiring the memory it will ensure that it will not be swapped to disc. Otherwise, static linking can be used.

There were two main changes to the protocol server. Firstly, the TCP control block (the per connection TCP state information) was modified to include the variables required to support dynamic protocol function selection. As the size of the TCP control block cannot be significantly expanded (because it must fit within an `mbuf` [81]), a new data structure was created. This data structure is shown in figure 4.5. The variables for the input and output data streams are duplicated so that the protocol functions used in either direction are independent. The comments adjacent to each variable describes its purpose.

A pointer to the `tcp-adapt` data structure was placed in the TCP control block. However, for each TCP control block, there is a unique instance of this data structure. That is, there is one (and only one) instance of this structure per TCP connec-

```
                /* the Error Control Function Pointer type */
typedef unsigned int (*tcpInEP)(register mbuf* m, register int pLen);

                /* the input Flow Control Function Pointer type */
typedef int (*tcpInFP)(
                register struct tcpiphdr* ti,
                register struct mbuf* m,
                register struct tcbcb* tp,
                struct inpcb* inp,
                struct mbuf* om,
                struct socket* so,
/* return values */
                int* needoutput,
                int* issP,
                int* dropsocketP
              );

                /* the output Flow Control Function Pointer type */
typedef int (*tcpOutFP)(register tcpcb* tcpControlBlock);

struct tcp_adapt
{
/* INPUT related variables */
  u_short    ta_lastFDec;        /* previous IN flow control decision */
  u_short    ta_noFlowCtrl;        /* prev. IN TCP segment flow ctrl flag*/
  tcpInFP    ta_inFFn;         /* input Flow Ctrl function ptr */

  u_short    ta_noEState;        /* previous input error control state */
  tcpInEP    ta_inEFn;         /* input error control function ptr */

/* OUTPUT related variables */
  u_char     ta_outNoCkSum;        /* previous output error control state*/
  u_char     ta_outNoFlowCtrl;     /* previous output flow control state */
  tcpInEP    ta_outEFn;         /* output error control function ptr */
  tcpOutFP   ta_outFFn;         /* output flow control function ptr */
};
```

Figure 4.5: *The Per Connection Data Structure (*`tcp-adapt`*) Used to Allow Dynamic Linking.*

tion. This was achieved by modifying the function `tcp-newtcpcb()` (which creates a new TCP control block) to also create an instance of `tcp-adapt`. The data structure is initialised so that none of the adaptive protocol functions are present.

At connection tear-down any linked functions are unlinked to free the associated memory. This was achieved by modifying the function `tcp-close()`. The normal purpose of this function is to remove the control block associated with a TCP connection.

The other main change to the protocol server was to access `tcp-adapt`. For the input side of the connection `tcp-input()` was replaced with a new function called `tcp-inputMgr()`. This function is responsible for configuring the input data stream. This function receives packets from the IP layer (upcalled from `ip-input()`), performs demultiplexing, and then determines if a change in the protocol functions is required (including changes from no protocol function). If there is, the appropriate functions are linked in. Finally, the protocol functions are called. The code to link in and execute error control is shown in figure 4.6.

It can be seen from figure 4.6 that there can only be two types of error control (the checksum is either on or off). This is specific to this implementation, and is addressed by the Universal Transport System §5.

Similar changes were made to the output functions of TCP (`tcp-output()` was replaced by `tcp-outputMgr()`, which then upcalls `ip-output()`). The changes made to the input and output data streams are independent so that the protocol functions used in either direction are also independent. Thus, for example, error control may be used for the input data stream, but not used for the output data stream of the same connection.

For the experiments, the switch from one mode of operation of TCP to another was initiated at the transmitter by the application when half the user data has been

```
/* link in Error Control if required */
if ((ti→ti_flags & TH_NOCKSUM) ≠ tAdapt→ta_noEState)
{                                       /* has error control changed? */
  tAdapt→ta_noEState = (ti→ti_flags & TH_NOCKSUM);

  if (tAdapt→ta_noEState)
  {                                     /* turn OFF error control */
    if (tAdapt→ta_inEFn)                        /* unlink if pointer in use */
      dld_unlink_file_count("tcp˙input˙con.o", 1);
    dld_link_count("tcp˙input˙coff.o");
    tAdapt→ta_inEFn = (tcpInEP) dld_get_func("tcp˙input˙coff");
  } else {                               /* turn ON error control */
    if (tAdapt→ta_inEFn)                        /* unlink if pointer in use */
      dld_unlink_file_count("tcp˙input˙coff.o", 1);
    dld_link_count("tcp˙input˙con.o");
    tAdapt→ta_inEFn = (tcpInEP) dld_get_func("tcp˙input˙con");
  }
}

/* perform Error Control */
if ((ti→ti_sum = (*(tAdapt→ta_inEFn))(pkt, pktLgth)) ≠ 0)
{                                       /* error control failed */
  goto drop;
}
```

Figure 4.6: *The Code Used by* `tcp-inputMgr()` *to Link in Error Control. pkt is a pointer to the incoming packet, while pktLgth is the size of the packet.*

passed to the socket interface. Further, the two reserved bits in the TCP flags field were used to indicate if error control, or flow control was to be performed on the packet. A frame structure and protocol design that permits the generalisation of this concept is presented in §5.

The decision as to whether to perform error control at the receiver is based purely on the value of the TCP flag used in the packet header. The decision as to whether to perform flow control is based on the flags of the current and previous packets. To enable this, a boolean variable (`t-noFlowCtrl`) was also added to the TCP control

block of a connection. This variable is set to the value of the TCP flow control flag of the previous packet. Flow control is performed if the flow control flag in the current TCP packet and `t-noFlowCtrl` are both true.

When switching off flow control at the transmitter, only the data currently acknowledged can be guaranteed to be received. If data that has been transmitted but not acknowledged is needed, the switch to no flow control must be delayed until the critical data is acknowledged.

At connection tear-down any linked functions had to be unlinked to free the associated memory. This was achieved by modifying the function `tcp-close()` whose function is to remove the control block associated with a TCP connection.

4.5.3 The Effect on the Code Size

Table 4.4 compares the size of an embedded conditional statement implementation of the protocol server with that of a statically and dynamically linked version of the protocol server.

The use of embedded conditional statements results in the lowest size executable. The statically linked version is 2.3% larger (4694 bytes).

However, the addition of dynamic linking has resulted in a substantially larger increase in the executable size. This is because the dynamic linking library (22486 bytes) must be included into the executable. The result is a 13% (28754 bytes) increase in executable size[3] compared to the statically linked version.

[3]This assumes that all the protocol functions are pre-loaded into the server.

Realisation	Function	Executable Size (bytes)
Embedded	Protocol Server	185192*
Static Linked	Protocol Server	189886*
Dynamic Linked	Protocol Server	196689*
	Error Control On	829
	Error Control Off	120
	Flow Control On (Tx)	4658
	Flow Control OFF (Tx)	4551
	Flow Control On (Rx)	6201
	Flow Control OFF (Rx)	5601
	TOTAL	218640

Table 4.4: *A Comparison of the Executable/Object File Sizes of Various Realisations of the Protocol Server on the i486 Platform.*
** The Protocol Server entries for the integrated and Statically linked versions includes all the protocol functions. The Synthesised protocol server does not include the listed protocol functions, but it does include the DLD package (22486 bytes). Without this overhead, the Synthesised Protocol Server would be 174203 bytes.*

4.6 Performance Results

The effect on data throughput was observed when flow control and error control functionality were modified during the lifetime of a connection. For the experiments it was assumed that both ends could support the desired protocol functions. As already discussed, this eliminates the need for a finite state machine to ensure the availability of the protocol functions.

In order to interpret the results, the instantaneous throughput was calculated. As shown by figure 4.7, throughput at the transmitter was calculated by dividing by the time from the *previous* packet. However, at the receiver, the time measured was to the start of the *next* packet to be received.

A typical graph showing the throughput when both error control and flow control are removed is given by figure 4.8. The results show that there is a regular oscillation

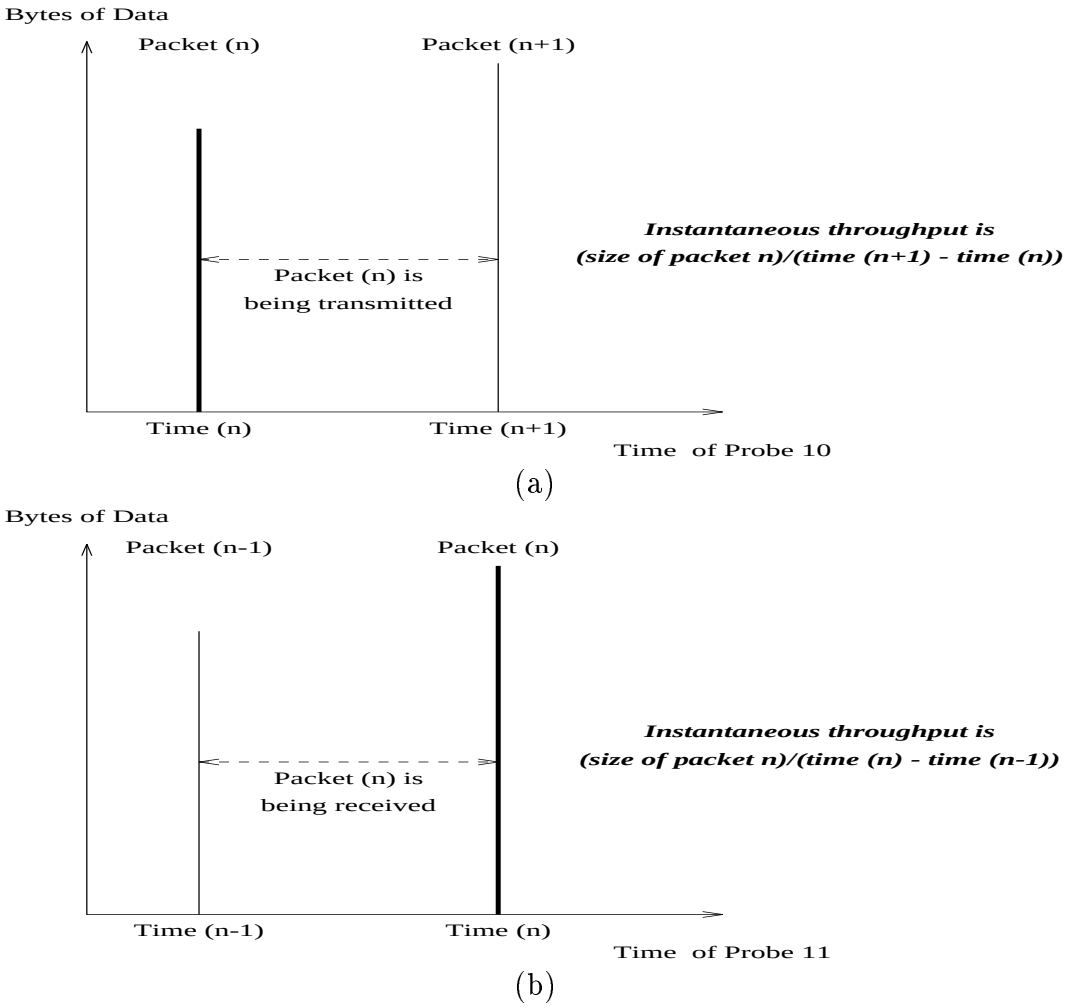


Figure 4.7: *An Illustration of the Method used to Calculate the Instantaneous Throughput at (a) the Transmitter and (b) the Receiver.*

in throughput. The drop in throughput is due to flow control (before the protocol function change) and the IPC interface between the application and the protocol server (see figure 3.5 for more details).

The graph shows that the change in throughput occurs instantaneously and smoothly when error control and flow control are switched off. The average steady state throughput, given by the solid line in the graphs, is the same magnitude as the static cases shown in tables 3.7 and 3.8.

When flow control is switched off (without modifying error control), a similar graph to figure 4.8 is produced. The effect of switching error control is that the instantaneous throughput between consecutive packets increases. However, the same cyclic pattern occurs irrespective of error control due to the flow control mechanisms.

For completeness, figure 4.9 shows the throughput when both error control and flow control are turned on again.

Before the switch, the receiver is operating under overload conditions. The transmitter has regular drops in throughput corresponding to data being transferred across the socket interface. (This effect is also present in figure 4.8 – but note that there is a different time scale for figures 4.8 and 4.9).

After the switch in protocol functions, congestion is detected which activates the slow start flow control mechanisms [43]. Thus the drop in throughput is due to the operation of both error control and flow control. The oscillations in throughput in this region are due to the limited congestion window at the transmitter.

Between 3 and 3.5 seconds ($3e{+}06$ and $3.5e{+}06$ μs) on figure 4.9a the congestion threshold was enlarged, allowing the throughput to increase.

4.7 Evaluating the Switching Time

A small drop in throughput during the switch is expected. At the transmitter, this is because of the following reasons:-

- processing an IPC message (approximately equal to the delay between probes 1 and 2 with 1 byte of data from table 3.3 – 897 μs),
- unlinking the old function from RAM (averaged from table 4.3 – 40 μs),

- linking the new from RAM (averaged from table 4.3 – 43 μs),
- return from the IPC (approximately equal to the delay between probes 19 and 20 with 1 byte of data from table 3.3 – 684 μs)

Further, the receiver must also perform dynamic linking from RAM.

The sum of the above delays gives a total overhead for adaptation, referred to as t_{adapt}, is 1664 μs. As the processor is busy performing the adaption, no data can be transmitted during this period.

Figure 4.10 shows an approximation of the instantaneous throughput during adaptation[4]. Region A represents the time that adaptation is performed (t_{adapt}). Region B represents the improved throughput due to the change in protocol functions. The time $t_{catchUp}$ represents the point where region A has the same area as region B. This is is the time that the protocol server has caught up to where it would have been had no adaption occurred. Beyond this time, the adaption will result in improved performance.

It can be shown that $t_{catchUp} = \frac{t_{adapt} \times th_2}{th_2 - th_1}$. This section has shown that $t_{adapt} = 1664\mu$s. Table 4.5 shows the calculated catch up time for various protocol function switches performed in this thesis. This value can be interpreted as the minimum time between adaptations that will permit an improved in performance. The last column of the table gives the maximum rate that adaptation can occur without an performance penalty.

These calculations have been based on the assumption that no finite state machine is required to confirm the availability of the protocol functions. If this was added, the processor would be required to perform extra processing,denoted by t_{fsm}.

[4]The time to perform the adaption (t_{adapt}) is small relative to the time scales on figures 4.8 and 4.9 and thus its effects cannot be seen on figures.

From	To	$t_{catchUp}$ (μs)	Max Switching Rate (Hz)
TCP	No Error Control	17675	57
TCP	No Flow Control	4309	232
TCP	No Error or Flow Control	4133	242
No Error Control	No Error or Flow Control	4887	205
No Flow Control	No Error or Flow Control	62295	16

Table 4.5: *The Calculated Catch Up Time for Various Adaptations. The throughput values used to calculate $t_{catchUp}$ were taken from the i486 column on table 3.7.*

Further, a three way handshake is required. This requires the protocol to operate with the original protocol functions for a round trip time (denoted by t_{rtt}). The handshake will occur after the new protocol functions are loaded from RAM.

From figure 4.11, it can be seen that now $t_{catchUp} = t_{rtt} + \frac{(t_{adapt}+t_{fsm})\times th_2}{th_2-th_1}$. Assuming that t_{fsm} is negligible, and taking the measured round trip time of the system, which was 7644μs, then the configuration in table 4.5 that has the lowest maximum switching rate (from No Flow Control to No Error or Flow Control) changes from 16 to 14 adaptations per second.

4.8 Conclusion

This chapter has shown that there is a need for protocols to be adaptive. A few tools have been outlined that can achieve this.

The use of dynamic linking was then investigated. It was shown that dynamic linking introduced a low overhead if the protocol functions were pre-loaded into RAM. However there was a large overhead involved in loaded the object file from disc making this approach unacceptable. Thus, if all the functions need to be pre-loaded, then effectively static linking is being performed. Thus the only benefit of

dynamic linking over static linking is that when new protocol functions are added to the system, the program does not need to be re-linked.

However, it was shown that the dynamic linker does increase the size of the executable file. Thus, unless there is a benefit in not being required to perform a linking operation at compile time, static linking is sufficient.

Adaptivity was then applied to the protocol server used in the previous chapter. The results confirmed that the protocol server's performance did not degrade with adaptivity being added to it.

It was shown that was a performance benefit only if the adaptation rate is no greater than at 14 Hz.

An unresolved issue has been highlighted, the effect on the protocol by the addition of mechanisms to ensure that both ends of a connection can support the desired protocol functions. However, it has been shown that dynamic protocol synthesis is both feasible, and efficient.

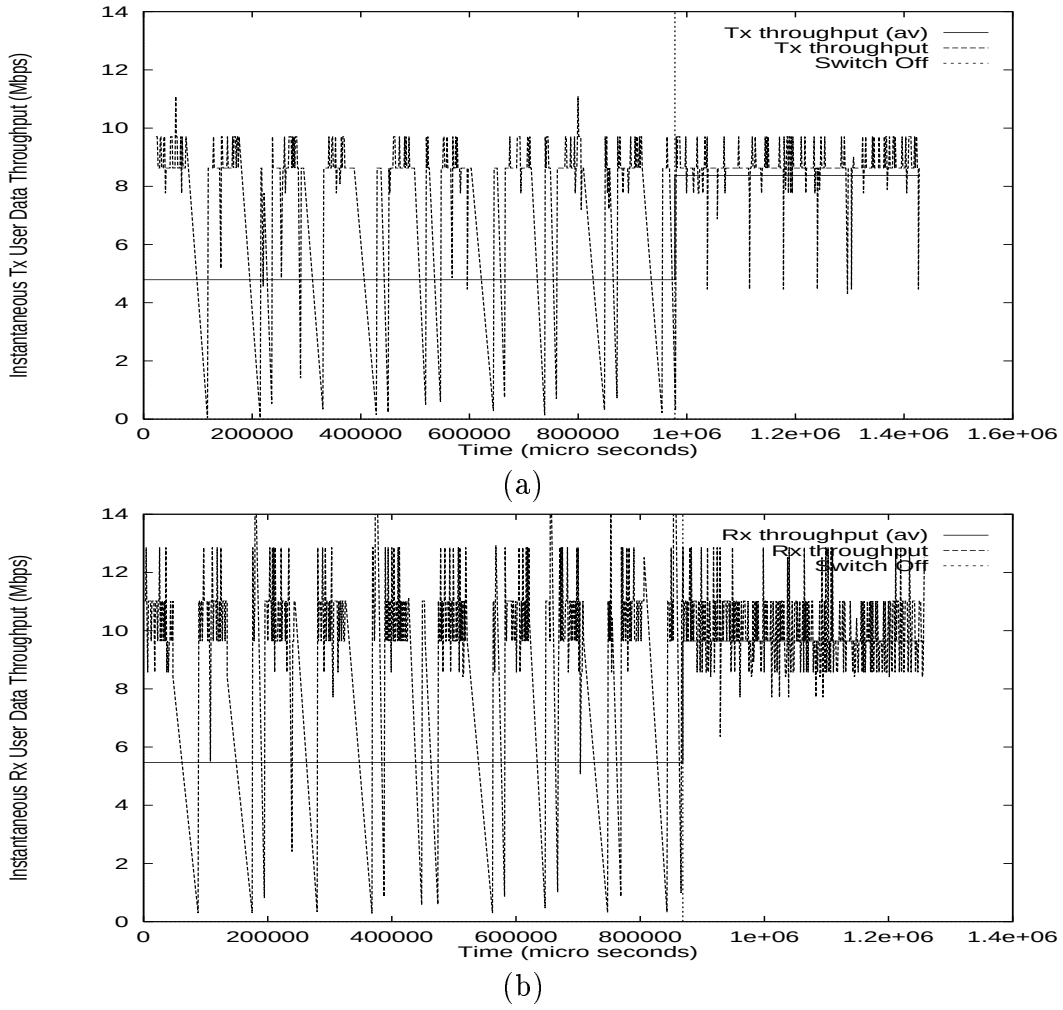


Figure 4.8: *The Instantaneous (a) Transmitter and (b) Receiver Throughput When both Flow and Error Control are Switched off at the Transmitter.*
Due to the resolution of the timer, it is possible for the instantaneous throughput to appear to be greater than 10Mbps. This is due to dividing by a short time interval. It can be shown that full packets travelling at 10Mbps can result in measurements up to 11Mbps. For small packets, and thus small transmission times, the measured time can be even greater.

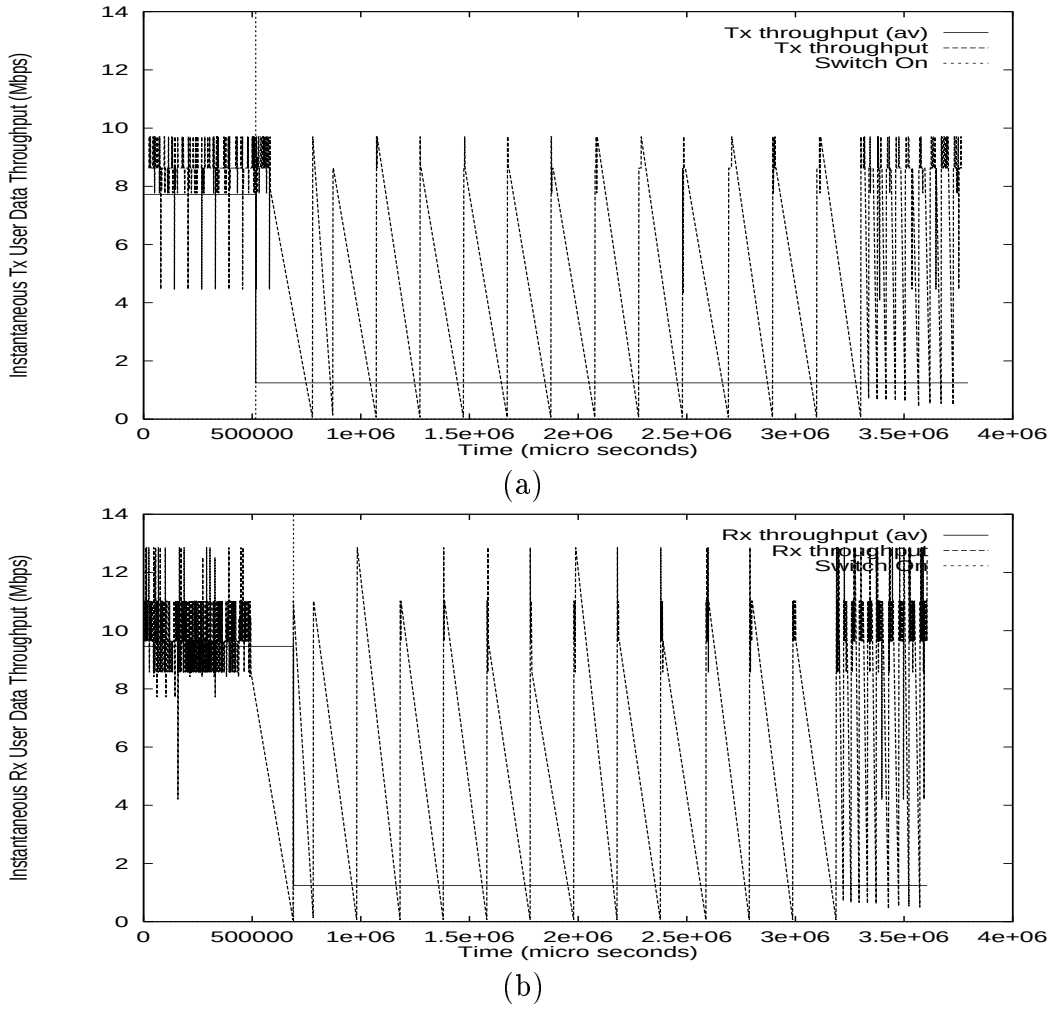


Figure 4.9: *The Instantaneous (a) Transmitter and (b) Receiver Throughput When both Flow and Error Control are Switched on at the Transmitter.*
Due to the resolution of the timer, it is possible for the instantaneous throughput to appear to be greater than 10Mbps. This is due to dividing by a short time interval. It can be shown that full packets travelling at 10Mbps can result in measurements up to 11Mbps. For small packets, and thus small transmission times, the measured time can be even greater.

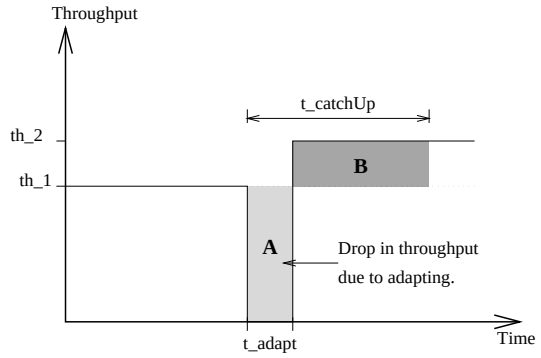


Figure 4.10: *An Approximation of the Throughput During Adaption.*

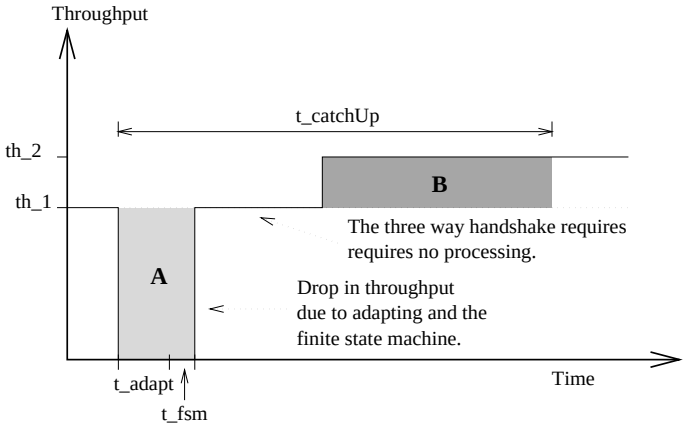


Figure 4.11: *An Approximation of the Throughput During Adaption When a Three Way Handshake is Used.*

Chapter 5

The Universal Transport System

The Universal Transport System (UTS) is a generalisation of the concepts presented in the previous chapters. Its design is based on the protocol decomposition given in §3.1 and the results of §3.2 which showed that significant gains can be achieved if unnecessary protocol functions are not performed. A need for adaptive protocols was illustrated by §4.1. Finally, §4.5 has shown that run time protocol synthesis is both readily implementable and requires only a small overhead during adaptation.

This chapter will discuss an adaptive protocol framework (corresponding to the middle layer of figure 4.1) and then will discuss some implementation issues related to this protocol.

5.1 Overall Operation

The conceptual architecture of the universal transport system, based on [93] and [25], is shown in Figure 5.1. It consists of a Functionality Selector, a Synthesis

Engine, a Protocol Function Library, and a Run Time Protocol.

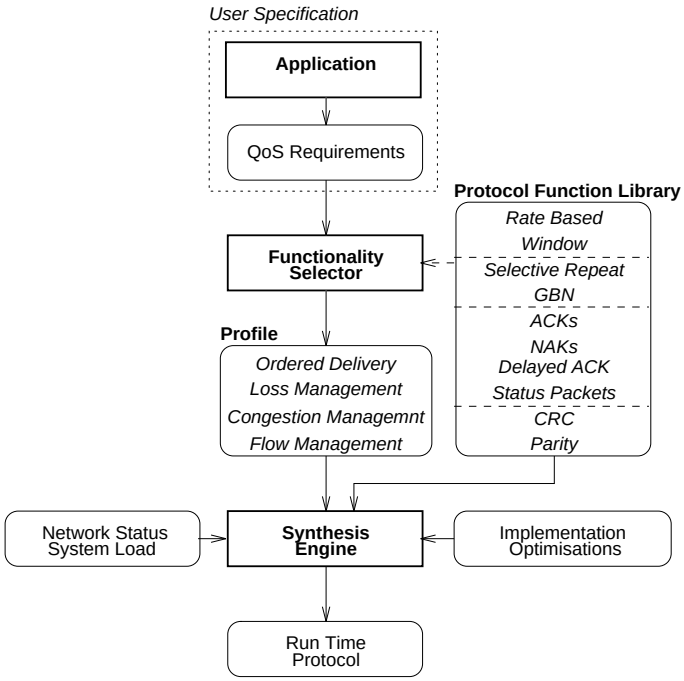


Figure 5.1: *The Conceptual Architecture of the Proposed Transport System.*

5.1.1 Initial Connection Establishment

An application will specify its QoS Requirements to the Functionality Selector. This can be done both during connection establishment and at any other time during the connection. The Functionality Selector then translates the application's QoS parameters into a set of required protocol functions, known as a profile. For example it will deal with decisions, such as whether Error Control is required. However, if a function is required it will not indicate *how* this functionality is supported.

The Profile is used by the Synthesis Engine to select the required functions from the Protocol Function Library. This selection will result is a particular realisation of a selected protocol function. For example, Error Control may be realised using a

Forward Error Correcting (FEC) scheme, or an Automatic Repeat-Request (ARQ) scheme.

The Protocol Function Library contains a complete collection of algorithms that can be used to implement the Protocol Functions. The dashed line joining the Library to the Functionality Selector indicates that the Functionality selector can only select Functions that can be realised by the Library.

The Synthesis Engine then creates a Run Time Protocol by combining the selected PFs from the PF library. To ensure efficiency, the Synthesis Engine takes into account ILP, Upcalls and other Implementation Optimisations[1].

5.1.2 Protocol Profile Negotiation

Two modes of operation are defined for the Synthesis Engine – connection establishment mode and adaptive mode. The connection establishment mode is used to initiate a connection by using only the protocol functions that are known to be available on all end systems.

After a connection has been established, the Synthesis Engine changes to adaptive mode. In this mode, it is possible for the Synthesis Engine, through the use of a three way handshake, to adapt the protocol. The Synthesis Engine will adapt the protocol in response to user requests, end system constraints, and variable services available from the underlying network.

[1] ILP needs to be investigated further and is an open research issue.

5.2 Operational Requirements

In order to design this system, the initial connection establishment, support for an adaptive protocol, and a means to incorporate new protocol functions need to be developed. This section will describe this support.

5.2.1 Initial Connection Management

Standard protocols (network wide) need to be used to establish a connection so that the it can be ensured that the selected protocol functions will be supported at an arbitrary host. This has been achieved through the use of *Mandatory* PFs – ones that are guaranteed to be available everywhere[2]. The Synthesis Engine can only use these functions whenever it is in connection establishment mode.

Connections are established using a three way handshake similar to TCP. When the handshake has completed, the Synthesis Engine changes from connection establishment mode to adaptive mode.

5.2.2 Adaptation Protocol

An error control scheme similar to XTP [83] will be used. Periodically the transmitter will send a control packet (known as a poll) requesting the current state of the receiver. When the receiver receives this packet it replies with its state information for the connection (known as a status packet).

This scheme was chosen because only one timer per connection is required to implement it.

[2] If a PF is not mandatory, then it is said to be optional.

5.2.3 Control Packets for Negotiation

To ensure reliability in adaptive mode, a three way handshake is used to change protocol functions, as summarised by figure 5.2 and illustrated by figure 5.3. This information will be piggy-backed onto the control packets described in the previous section.

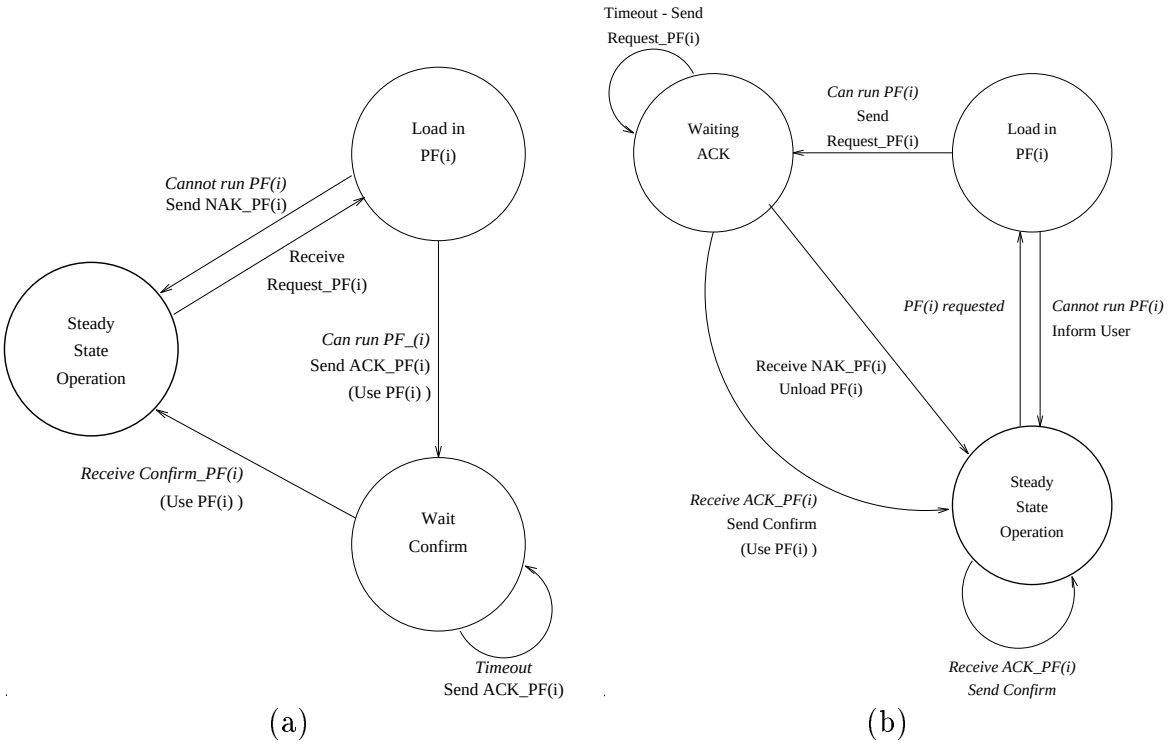


Figure 5.2: *The State Transition Diagram for the Synthesis Engine. (a) is for the side that responds to the change while (b) is for the side that initiates the change.*

During steady state operation in the adaptive mode, all the chosen protocol functions are in use.

The SE at either the transmitter or receiver can initiate a change of protocol functions. If the SE (at side B) decides to initiate a change of the optional PFs, it sends a request for a new or changed PF. For mandatory PFs, the SE can request it

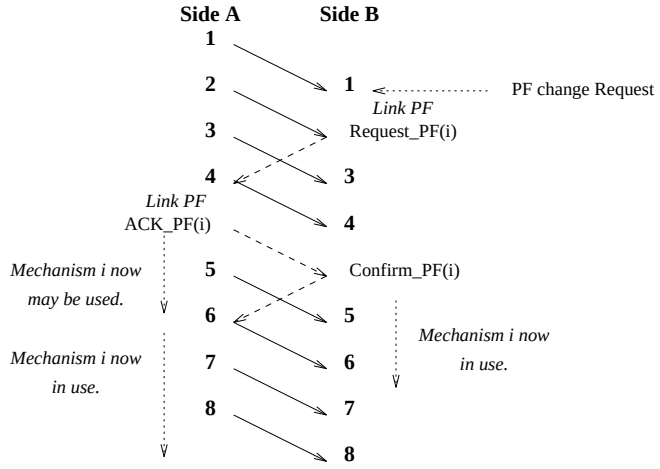


Figure 5.3: *An Illustration of the Operation of the Synthesis Engine.*

without any overhead to confirm the availability and usability of the PF. Otherwise the SE must first confirm that the PF is available.

The SE at side A receives this request, and changes from steady state operation to the Load in PF state. Then either the requested PF will be rejected (a NAK is transmitted) or it will be accepted and an ACK is sent. The SE, while waiting for the confirmation message, will carry out the requested PF operations. The state diagram shown by figure 5.2 illustrates the protocol's response to lost control packets.

When the ACK message is received at side B, it confirms that the new or changed PF is now in use, and SE replies with a confirmation. The SE goes back to steady state operation.

When the SE at A receives the confirmation and also returns to steady state operation. If the original request was to change a PF, then the original is no longer accepted.

If a requested protocol function is not available at the remote end, as shown by figure 5.2, a `NAK-PF(i)` packet is returned.

The three way handshake of figure 5.3 ensures reliability whenever the control packets are lost. If the `Request-PF(i)` packet is lost, then a timer at side B (which is waiting for the `ACK-PF(i)` will timeout and re-send the `Request-PF(i)` packet. If the `ACK-PF(i)` is lost then either, depending on which event occurs first, the timer at side B will timeout and send another `Request-PF(i)` or the timer at side A (which is waiting for the `Confirm-PF(i)` will timeout and the `ACK-PF(i)` will be resent. Finally, if the `Confirm-PF(i)` is lost, then the timer at side A will timeout and the `ACK-PF(i)` will be resent.

5.3 Structures for Supporting Adaptation

In the experiments carried out to prove the concepts presented in §4.5; the reserved bits in the TCP header were used. However, to generalise this concept, it is necessary for the protocol's structure to be designed to support protocol mechanism selection.

The proposed Universal Transport System (UTS) implementation will operate over IP. IP was selected as it has very low processing overheads; this has been shown both in this thesis and elsewhere [21]. Further, it performs all the requirements of a point to point protocol. A new prototype value in IP will be used to select a UTS packet.

However, the frame format of the higher layer protocols will differ from existing systems. The decision not to use the existing architectures was made because they do not efficiently allow enough flexibility, scalability and adaptivity.

In particular, TCP has a very limited header format. Without using the options field, this does not allow new protocol functions to be added. The use of options is limited because (a) it does not readily support a three way handshake when the protocol adapts, and (b) the new protocol function fields must be placed after the

standard header. Thus the order of the protocol function fields cannot be optimised to allow efficient protocol processing. Further, trailer fields do not exists which prevents optimisations such as placing the checksum calculation at the end of the data.

The XTP [83] frame format allows greater flexibility than that of TCP as various protocol functions can be configured at the time of compilation, but mid-stream adaptation (through a three way handshake) is not supported. However, aspects of XTP's design are included in UTS, such as separating data and control packets and using a *poll-status* paradigm. That is, the transmitter sends periodic poll packets to the receiver, which replies with its current status[3]. These packets can also carry user data, allowing request-response applications to be efficiently supported. This paradigm allows error control to be implemented using only one timer per connection [30, 60].

5.3.1 Frame Format

To allow flexibility, the format of a UTS packet consists of a series of protocol functions. A fixed header format was not selected because it could not be guaranteed to support all future protocol functions. Figure 5.4 shows the frame format proposed for UTS. The end of the header list is indicated by a PF length of 0.

Figure 5.4 also shows the format of a protocol function. The fields are:-

PF Length is the length of the PF field (in multiples of 32 bits), as shown by figure 5.4. Variable length options are required so that no limitation is placed.

[3]The period between transmitting the state (control) information (and thus the period of this timer) determines the rate that the transmitter is updated with the status of the receiver, and thus the rate that the transmitter can free resources. This limitation places a lower-bound on the period of the timer, which in turn places a lower-bound on its processing overhead.

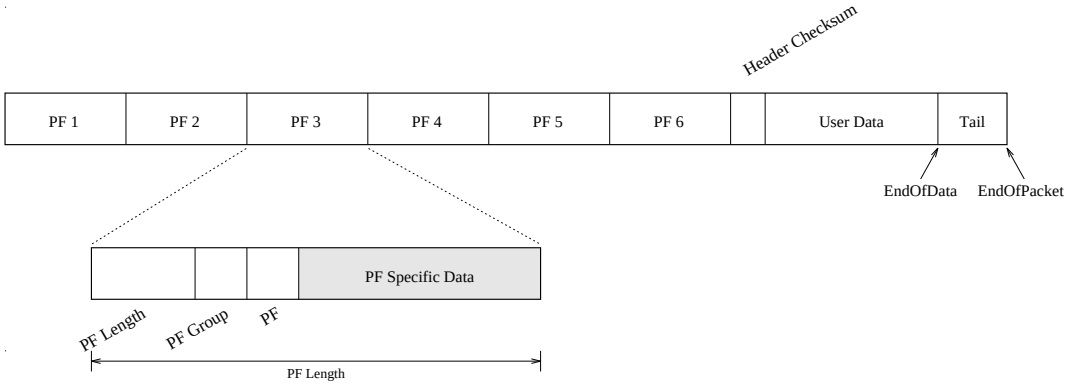


Figure 5.4: *An Example of a UTS header.*
The pointers illustrated are part of the Synthesis Engine.

PF Type is the type of PFs, such as error control, flow control, etc.

PF specifies which mechanism is being used to realise the PF Type, for example, *checksum* or *forward error correction* for error control.

The `PF Length` field will have a fixed length of 32 bits, while the `PF Type` and `PF` fields will have a combined length of 32 bits.

The following variables will be maintained by the Synthesis Engine (and visible to the protocol functions):-

- `endOfData` (end of user data in the packet),
- `endOfPacket` (the difference between this, and `endOfData` allows space for error control, etc.).

When a new packet arrives at the Synthesis Engine, then `endOfData` and `endOfPacket` are set to the last byte of the packet. If a protocol function (such as Error Control) has data at the tail of the packet then the location currently pointed to by `endOfData` is used by the protocol function. The protocol function must also move `endOfData` to reflect the number of bytes it used in the tail. Thus, after all the

protocol functions have been performed, `endOfData` will point to the true end of the data field.

5.3.2 Adaptation Mapping

A flexible method is required to allow the protocol to map protocol functions to numbers used by the control packets. A method similar to TCP/UDP port numbers is proposed; a file (controlled by the system administrator) is read by the configuration engine to give this mapping.

This architecture allows new optional protocol functions to be created by including the operation of the function in the protocol function library. The system administrator will ensure that this mapping is unique.

Tables 5.1 and 5.2, which are based on figure 3.1, give an example of how the above concept is to be realised.

PF Group	Numerical Mapping
Error Control	0
Flow Control	1
Sequencing	2
Connection Management	3

Table 5.1: *An Example of the PF Groups.*

PF	Numerical Mapping	Mandatory
16 bit Checksum Calculation	0	Yes
32 bit Checksum Calculation	1	Yes
Forward Error Control - Hamming	2	Yes
Forward Error Control - Reed-Solomon	3	No
64 bit Checksum Calculation	4	No

Table 5.2: *An Example of the Protocol Functions for Error Control.*

Chapter 6

Conclusions

This thesis has successfully shown that a protocol can be factorised into its component protocol functions. It was shown that a common set of protocol functions was required irrespective of the type of end-to-end protocol.

Experiments showed that when these protocol functions are fully recombined, the resulting protocol is as efficient as the original integrated protocol. However, it has been shown that there are performance benefits resulting from creating a protocol with reduced functionality. Further, this approach allows the functionality provided by a protocol to be matched to the requirements of a given application.

The need to adapt a protocol during a data transfer was then identified. An implementation of an adaptive protocol showed that it was feasible and practical to perform adaptation when it was known that both ends of the connection could support the desired protocol functions.

It was shown that the time delays associated with dynamic linking (and in particular the time required to load the protocol functions from disc) was unacceptable.

Thus the protocol functions need to be pre-loaded into the computer's memory to remove this overhead. This is equivalent to using static linking.

Further, it was shown that the use of a dynamic linker in this fashion resulted in greater memory use than a statically linked version of the program. Thus, despite its advantages such as extendibility, the use of dynamic linking in its current form does not provide any performance advantages.

Adapting the protocol functions during a connection required a small processing overhead. This overhead did not adversely affect the throughput of the protocol. The minimum time between permitted adaptations was also calculated.

It has been shown that adaptivity provides benefits to the application. That is, given the additional overhead, the use of adaptivity within a protocol is advocated.

A general framework was then developed to allow adaptivity to be introduced into the protocols.

Finally it was shown that the platform used affected the performance of the protocol. A 486 (CISC) processor was less sensitive to the data touching operations (such as error control) compared to a Dec 5000/240 (RISC) machine. This is because the difference in processor and memory speeds are greater on RISC processors than on CISC ones. Thus RISC processors are more dependent on their memory caches than CISC machines are.

6.1 Critical Discussion

The experiments presented in this thesis have mainly concentrated on a transport service. This needs to be expanded to include factorising and performance experiments involving the whole protocol stack, possibly including some of the common

application functions such as decoding.

It has been shown in literature that protocols executed by an application have performance benefits over a protocol executed by the kernel of an operating system or by a protocol server. The experiments in this thesis were performed using a protocol server.

An adaptive interface has been discussed in this thesis, but a formal interface has not been designed for the protocol. The current interface uses the same IPC interface as the data and involves parsing a string that was passed to the protocol server from the application.

The experiments in chapter 4 assumed that the protocol functions were available at both end systems. Thus protocol developed did not need to check that the selected protocol functions could be understood by the other end of the connection. Analysis speculated at the effect this extra overhead would have, but it was not confirmed with experiments.

An adaptive framework for unicast connections that includes a three way handshake to ensure the availability of the selected functions was developed. However, no experiments were performed using this framework, nor has it been compared with an implicit adaption request scheme. Further, the design only supports unicast connections.

It has been shown in literature that Integrated Layer Processing (ILP) is required for efficient protocol implementations. A factorised protocol is performing the reverse by separating the protocol functions. A scheme is required that will allow the various protocol functions to be specified in isolation, but will result in an integrated realisation of the data manipulation functions.

This is still an open issue, however it is speculated that ILP can be incorporated

into the protocol as follows. Whenever a packet is received or sent it is processed in four stages [2].

- ensure the integrity of the header,
- perform the preliminary processing of the PFs,
- perform the ILP loop,
- perform the final processing of the PFs.

Application specific functions, such as the presentation layer functions, are incorporated into the ILP loop. Two ways are envisaged at present to implement this loop dynamically, either by using self-modifying code [53, 58] or by using dynamic linking. If self-modifying code is used, then an ILP template will be created per connection. The data manipulations are then copied into the loop as required. Alternatively an ILP Loop can be compiled per connection (using similar techniques to the static case [9]) and then dynamically linking it into the protocol server [90].

Finally, this thesis has looked at **how** to adapt, not **when** to adapt. Heuristics need to be developed to guide adaptation. Further, an interface needs to be designed between an adaptive protocol system and the network management system to facilitate the adaptation process.

6.2 Future Directions

It can be seen from the previous section that further work is required. This includes:-

- developing and experimenting with a factorisation of the whole protocol stack,

- investigating the benefits of moving the protocol server into the user space of each application,
- developing a generic adaptive interface to the system,
- comparing the benefits of a three way handshake with an implicit scheme to allow reliable adaptivity,
- investigating the effects on performance of a three way handshake, and an implicit scheme, which ensures reliability when adapting,
- developing a framework for multicast connections,
- incorporating an ILP loop into the system,
- developing heuristics on when to adapt, and
- developing an interface to network management.

Appendix A

A Statistical Analysis of the Timer Resolution

The timer used for all the experiments had a resolution of 156μs. This resolution is very coarse relative to the measurements. This appendix analyses the errors introduced by this resolution, and concludes that for the experiments involving 400 repetitions, the error introduced by the timer can be approximated by a random variable with a normal distribution, a mean of 0μs, and a standard deviation no greater than 3.9μs.

This analysis, which uses formulas from [45], is composed of two steps. First consider figure A.1, which illustrates an interval of time less than or equal to the resolution of the clock.

One of two outcomes can occur when timing this event. If no timer boundaries occur within the timed interval (r), then the measured time is 0, with a probability of $1 - P = \frac{r-y}{r}$. Otherwise, the timer boundary will occur within the timed interval, in which case the measured time is r, with a probability of $P = \frac{y}{r}$. When this

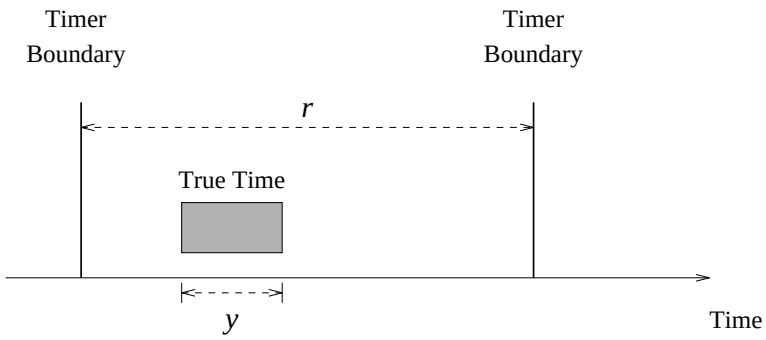


Figure A.1: *The Relationship Between a Time Interval less than or equal to the resolution of the Timer.*

experiment is repeated, this Binomial distribution tends to a Normal Distribution. With N repetitions, the mean is $\mu_N = \frac{Ny}{r}$. Converting this into the quantity of interest gives $\overline{y} = \frac{\mu_N \times r}{N}$.

The variance of the binomial distribution is given by $NP(1-P)$. Thus the variance of $\overline{y}$ is $\sigma^2 = \frac{r^2 P(1-P)}{N}$. Substituting for P gives $\sigma^2 = \frac{y(r-y)}{N}$. ($y$ varies from 0 to r).

The resolution of the clock (r), and the number of experiments (N) are fixed. Thus σ^2 is given by a concave down parabolic function of y. This has a maximum value of $\frac{r^2}{4N}$ when $y = \frac{r}{2}$.

For the experiments used is this thesis, $r = 156\mu$s and $N = 400$. Thus the maximum variance of the measurements (due to the resolution of the clock) is $15.21(\mu s)^2$. This is a standard deviation of 3.9μs.

Step 2 of this analysis involves intervals greater than the resolution of the clock. For these intervals, the true period being timed can be broken into 2 regions, region A being the maximum multiples of the resolution of the clock, and region B being the rest of the interval being timed (which is less than the resolution of the clock), as shown in figure A.2.

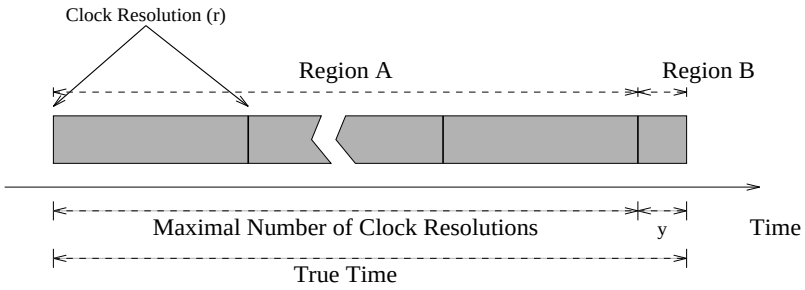


Figure A.2: *Timed Intervals Greater than the Resolution of the Clock.*

As region A is a multiple of the resolution of the clock, the measured time will be the same as the true value no matter where the interval starts with respect to the clock. (That is, the mean of measuring region A is the true value of region A, and the standard deviation is zero). Region B satisfies the requirements for the previous analysis, giving a mean equal to the true mean of region B (over N samples) with a maximum variance of $\frac{r^2}{4N}$.

When combining these regions, as already noted, the variance of region A is zero irrespective of when this interval starts with respect to the clock. Thus the variance of the overall region is that of B. This is $\frac{r^2}{4N}$, or $15.21(\mu s)^2$ for the experimental setup used. Again this corresponds to a standard deviation of 3.9μs.

Bibliography

[1] M. B. Abbot and Peterson, "A language based approach to protocol implmentation," *IEEE/ACM Transactions on Networking*, September, 1992.

[2] M. B. Abbott and L. L. Peterson, "Increasing network throughput by integrating protocol layers," tech. rep., Department of Computer Science; University of Arizona, 1992. Also appeared in IEEE/ACM Transactions on Networking.

[3] H. Absaloms, "Timer server implementations in computer communication protocols," Master's thesis, University of Technology; Sydney, July, 1992.

[4] P. D. Amer, T. J. Connolly, C. Chassot, and M. Diaz, "Partial order transport service for multimedia applications: Unreliable service," in *International Networking Conference*, August, 1993.

[5] G. Berry and G. Gonthier, "Incremental development of an HDLC protocol in esterel," tech. rep., INRIA, May, 1989.

[6] B. N. Bershad, "The increasing irrelevance of IPC performance for microkernel-based operating systems," tech. rep., Carnegie Mellon University, March, 1992. http://www/dml/local/docs/mach/IPCperf.ps.gz.

[7] J. Boykin, D. Kirschen, A. Langerman, and S. LoVerso, *Programming Under Mach*. Unix and Open Systems Series, Addison-Wesley Publishing Company, 1993. ISBN 0-201-52739-1.

[8] P. Boyle, "Wireless LANs: No strings attached," *PC Magazine*, pp. 215–237, January, 1995.

[9] T. Braun and C. Diot, "Protocol implementation using integrated layer processing," in *Sigcomm*, 1995. submitted.

[10] Z. L. Budrikis, "Lecture 3: Atm network resource mangement - connection admission control." CSIRO Workshop, June, 1995.

[11] R. Càceres and L. Iftode, "The effects of mobility on reliable transport protocols," *IEEE*, pp. 12–20, 1994.

[12] C. Castelluccia and W. Dabbous, "Modular communication subsystem implementation using an synchronous approach," tech. rep., INRIA, 1994.

[13] CCITT, "X214: Transport service definition for open systems interconnection for CCITT applications," Geniva, 1984.

[14] CCITT, "X224: Transport protocol specification for open systems interconnection for CCITT applications," Geniva, 1984.

[15] D. R. Cheriton and C. L. Williamson, "VMTP as the transport layer for high-performance distributed systems," *IEEE Communications Magazine*, pp. 37–44, June, 1989.

[16] G. Chesson, "XTP/PE design considerations," pp. 27–33, IFIP WG 6.1/WG 6.4, Elsevier Science Publishers B.V. (North-Holland), May, 1989.

[17] D. Clark, "The structuring of systems using upcalls," in *Proceedings of the* 10^{th} *ACM Symposium on Operating Systems Principles*, ACM, December 1-4, 1985.

[18] D. Clark, V. Jacobson, J. Romkey, and H. Salwen, "An analysis of TCP processing overhead," *IEEE Communications Magazine*, vol. 27, no. 6, pp. 23–29, June, 1989.

[19] D. Clark, M. Lambert, and L. Zhang, "NETBLT: A high throughput transport protocol," in *Proceedings of the SIGCOMM'87 Workshop on Frontiers in Computer Communications Technology*, pp. 353–359, August 11-13, 1987.

[20] D. Clark, S. Shenker, and L. Zhang, "Supporting real-time applications in an integreated services packet network: Architecture and mechanism," *ACM Computer Communication Review (SIGCOMM)*, vol. 22, no. 4, pp. 14–26, 1992.

[21] D. Clark and D. Tennenhouse, "Architectural considerations for a new generation of protocols," in *Proceedings of SIGCOMM'90, Communications Architectures and Protocols*, pp. 200–208, September, 1990.

[22] D. E. Comer and D. L. Stevens, *Internetworking with TCP/IP - Volume II - Design; Implementation; and Internals.* Prentice Hall, 1991. ISBN 0-13-472242-6.

[23] J. Crowcroft, I. Wakeman, Z. Wang, and D. Sirovica, "Is layering harmful?," *IEEE Network Magazine*, January, 1992. p. 7. 16 July 1991.

[24] A. Danthine, Y. Baguette, G. Leduc, and L. Léonard, "The OSI 95 connection-mode transport service – the enhanced QoS," in *High Performance Networking 92*, pp. E2.1–E2.18, IFIP, 1992.

[25] R. De Silva, L. Dairaine, A. Richards, A. Seneviratne, and M. Fry, "Automatic generation of dynamically adaptable protocols," in *Upper Layer Protocols; Architectures and Applications*, December, 1995. submitted.

[26] "Digital's DECstation family performance summary," Tech. Rep. EC-N0331-19, Digital Equipment Corporation., June, 1993. http://www.digital.com/home.html.

[27] M. Diaz, K. Drira, A. Lozes, and C. Chassot, "Definition and representation of the quality of service for multimedia systems," in *High Performance Networking*, 1995. submitted.

[28] C. Diot, I. Chrisment, and A. Richards, "Application level framing and automated implementation," in *High Performance Networking*, 1995.

[29] W. A. Doeringer, D. Dykeman, M. Kaiserwerth, B. W. Meister, H. Rudin, and R. Williamson, "A survey of light-weight transport protocols for high-speed networks," *IEEE Transactions on Communications*, vol. 38, no. 11, pp. 2025–2038, November, 1990.

[30] B. T. Doshi and P. K. Johri, "Communication protocols for high speed packet networks," *Computer Networks and ISDN Systems*, vol. 24, pp. 243–273, 1992.

[31] R. Draves, "A revised ipc interface," tech. rep., Carnegie Mellon University, 1991. http://www.ee.uts.edu.au/dml/local/docs/mach/ipc.ps.gz.

[32] P. Druschel, M. B. Abbott, M. A. Pagels, and L. L. Peterson, "Network subsystem design: A case for an integrated data path," *IEEE Network (Special Issue on End-System Support for High Speed Networks)*, vol. 7, no. 4, pp. 8–17, July, 1993.

[33] P. Druschel, L. L. Peterson, and N. C. Hutchinson, "Lipto: A dynamically configurable object-oriented kernel," *IEEE Technical Committee on Operating Systems and Application Environments.* ftp://ftp.cse.ucsc.edu/pub/tcos/v#n#.

[34] L. Fedaoui, A. Seneviratne, A. Fladenmuller, and E. Horlait, "Implementation of an end-to-end quality of service management scheme," in *High Performance Computing Conference*, September, 1994.

[35] E. W. Felton, "The case for application-specific communication protocols," Tech. Rep. 92-03-11, Department of Computer Science and Engineering; University of Washington, March, 1992. http://www.cs.washington.edu/research-/tr/techreports.html.

[36] G. H. Forman and J. Zahorjan, "The challenges of mobile computing," *IEEE Computer*, vol. 27, no. 4, pp. 38–47, April, 1994.

[37] M. Fry, A. Richards, and A. Seneviratne, "Framework for implementing the next generation of communication protocols," in *Fourth International Workshop on Network and Operating System Support for Digital Audio and Video*, 1993. Lancaster; England. ftp://ftp.ee.uts.edu.au/fry.daav93.ps.gz.

[38] T. Ginige, "Performance evaluation of standardised transport protocols for high speed data communications," Master's thesis, University of Technology; Sydney, September, 1992.

[39] P. Gunningberg, C. Partridge, T. Sirotkin, and B. Victor, "Delayed evaluation of gigabit protocols," in *Second MultiG Workshop*, 1991.

[40] F. Halsall, *Data Communications; Computer Networks and Open Systems.* Addison-Wesley Publishing Company, third ed., 1992. ISBN 0-201-56506-4.

[41] P. Hoschka, "Towards tailoring protocols to application specific requirements," in *Inforcom*, pp. 647–653, IEEE, 1993.

[42] N. C. Hutchinson, S. Mishra, L. L. Peterson, and V. T. Thomas, "Tools for implementing network protocols," *Software-Practice and Experience*, vol. 19, no. 9, pp. 895–916, September, 1989.

[43] V. Jacobson, "Congestion avoidance and control," *SIGCOMM'88*, pp. 314–329, 1988.

[44] V. Jacobson, "VAT - X11 based audio conference tool." UNIX Manual Pages, February, 1993.

[45] R. A. Johnson, *Miller and Freund's Probability and Statistics for Engineers*. Prentice-Hall Inc. Englewood Cliffs, NJ, 1994. ISBN 0-13-721408-1.

[46] J. Kay and J. Pasquale, "Measurement analysis and improvement of UDP/IP throughput for DECstations 5000," in *USENIX*, 1993. ftp://ftp.ucsd.edu/pub-/csl/fastnet.

[47] D. Keppel, S. J. Eggers, and R. R. Henry, "A case for runtime code generation," tech. rep., University of Washington, 1991. http://www.cs.washington.edu-/research/tr/techreports.html.

[48] B. W. Kernighan and R. Pike, *The UNIX Programming Environment*. Prentice-Hall Inc. Englewood Cliffs, NJ, 1984.

[49] T. F. La Porta and M. Schwartz, "Architectures, features, and implementation of high-speed transport protocols," *IEEE Network Magazine*, pp. 14–22, May, 1991.

[50] K. A. Lantz, W. I. Nowicki, and M. M. Theimer, "An empirical study of distributed application performance," *IEEE Transactions on Software Engineering*, vol. SE-11, no. 10, pp. 1162–1173, October, 1985.

[51] J.-Y. Le Boudec, "The asynchronous transfer mode: a tutorial," *Computer Networks and ISDN Systems*, vol. 24, pp. 279–309, 1992.

[52] S. J. Leffler, M. K. Mc Kusick, M. J. Karels, and J. S. Quaterman, *The Design and Implementation of the 4.3 BSD Operating System.* Addison-Wesley Publishing Company, 1989. ISBN 0-201-06196-1.

[53] A. Leone and P. Lee, "Lightweight run-time code generation," *ACM SIGPLAN Workshop on Partial Evaluation and Semantics-Based Program Manipulation*, June, 1994.

[54] S. Leue and P. Oechslin, "Formalizations and algorithms for optimized parallel protocol implementation," 1994.

[55] R. C. V. Macario, ed., *Personal and Mobile Radio Systems.* Peter Peregrinus Ltd., 1991.

[56] C. Maeda and B. N. Bershad, "Networking performance for microkernels," tech. rep., Carnegie Mellon University, March, 1992. ftp://mach.cs.cmu.edu/doc-/published/netperf.ps.

[57] C. Maeda and B. N. Bershad, "Protocol service decomposition for high-performance networking," tech. rep., Carnegie Mellon University, 1993. ftp:-//mach.cs.cmu.edu/doc/published/user.level.protocols.ps.

[58] H. Massalin, *Synthesis: An Efficient Implementation of Fundamental Operating System Services.* PhD thesis, Columbia University, 1992.

[59] J. Nagle, "Congestion control in IP/TCP networks," *ACM Computer Communications Review*, vol. 144, no. 4, pp. 11–17, 1984.

[60] A. N. Netravali, W. D. Roome, and K. Sabnani, "Design and implementation of a high-speed transport protocol," *IEEE Journal on Selected Areas in Communications*, vol. 38, no. 11, pp. 2010–2024, November, 1990.

[61] B. D. Noble, M. Price, and M. Satyanarayanan, "A programming interface for application-aware adaptation in mobile computing," in *Second USENIX Symposium on Mobile and Location Independent Computing*, April, 1995.

[62] S. O'Malley and L. L. Peterson, "A dynamic network architecture," *ACM Transactions on Computer Systems*, vol. 10, no. 2, pp. 110–143, May, 1992.

[63] On The Move Consortium, "Acts project proposal of the on the move consertium - mobile multimedia information services," March, 1995. Proposal Number 0092.

[64] M. A. Pagels, P. Druschel, and L. L. Peterson, "Cache and TLB effectiveness in processing network I/O," Tech. Rep. TR 94 08, Department of Computer Science - University of Arizona, March, 1994.

[65] C. Papadopoulos, *PhD Chapter 4: Unix Background and Probe Design*. PhD thesis, Washington University, 1993. ftp://dworkin.wustl.edu/pub/tcp.

[66] C. Papadopoulos and G. M. Purulkar, "Experimental evaluation of SUNOS IPC and TCP/IP protocol implementation," *IEEE/ACM Transactions on Networking*, vol. 1, no. 2, pp. 199–216, April, 1993.

[67] J. Postel, "File transfer protocol," *rfc 765*, June, 1980.

[68] K. K. Ramakrishnan and R. Jain, "A binary feedback scheme for congestion avoidance in computer networks," *ACM Transactions on Computer Systems*, vol. 8, no. 2, pp. 158–181, 1990.

[69] K. K. Ramakrishnan and H. Yang, "Interfacing to high speed networks." Tutorial - IEEE Local Computer Networks Conference, October, 1994.

[70] "Transmission control protocol - DARPA internet program protocol specifications," tech. rep., Information Sciences Institute - University of Southern California, September, 1981. ftped from rfc/TCP.Z at munnari.oz.au.

[71] A. Richards, R. De Silva, A. Fladenmuller, A. Seneviratne, and M. Fry, "The performance of configurable protocols," *Journal of High Speed Networks*, 1995.

[72] A. Richards, T. Ginige, A. Seneviratne, T. Buczkowska, and M. Fry, "An adaptive transport service suitable for high speed networks," *Concurrency: Practice and Experience*, vol. 6, no. 4, pp. 357–373, June, 1994. ftp://ftp.ee.uts.edu.au-/pub/prose/richards.concurrency94.ps.gz.

[73] A. Richards, A. Seneviratne, M. Fry, and V. Witana, "Tailoring the transport protocol for giga bit networks," in *Australian Telecommunication Networks and Applications Conference*, December, 1994. ftp://ftp.ee.uts.edu.au/pub/prose-/richards.atnac94.ps.gz.

[74] D. C. Schmidt, D. F. Box, and T. Suda, "ADAPTIVE: A flexible and adaptive transport system architecture to support lightweight protocols for multimedia applications on high-speed networks," in *Proceedings of the Symposium on High Performance Distributed Computing*, September, 1992.

[75] D. C. Schmidt, *An Object-Oriented Framework for Experimenting with Alternative Process Architectures for Parallelizing Communication Subsystems.* PhD thesis, Uiversity of California - Irvine, 1994.

[76] M. Siegel, M. Williams, and G. Röβler, "Overcoming bottlenecks in high-speed transport systems.," 16^{th} *IEEE Conference on Local Computer Networks*, October, 1991.

[77] K. R. Sollins, "The TFTP protocol (revision 2)," *rfc 783*, June, 1981.

[78] R. Steinmetz and C. Engler, "Human perception of media synchronization," Tech. Rep. 93.9310, IBM European Networking Center, 1993.

[79] W. R. Stevens, *UNIX Network Programming*. Prentice-Hall Inc. Englewood Cliffs, NJ, 1990. ISBN 0-13-949876-1.

[80] W. R. Stevens, *TCP/IP Illustrated; Volume 1: The Protocols*. Professional Computing Series, Addison-Wesley Publishing Company, 1994.

[81] W. R. Stevens and G. R. Wright, *TCP/IP Illustrated; Volume 2: The Implementation*. Professional Computing Series, Addison-Wesley Publishing Company, 1995.

[82] H. S. Stone, *High-Performance Computer Architecture*. Addison-Wesley Pubishing Company, second ed., 1992.

[83] W. T. Strayer, B. J. Dempsey, and A. C. Weaver, *XTP: The Xpress Transfer Protocol*. Addison-Wesley Publishing Company Inc. Reading; Massachusetts, 1992. ISBN 0-201-56351-7.

[84] C. A. Thekkath, T. D. Nguyen, E. Moy, and E. D. Lazowska, "Implementing network protocols at user level," *SIGCOMM*, pp. 64–73, 1993.

[85] Y. H. Thia and C. M. Woodside, "High-speed OSI protocol bypass algorithm with window flow control," May, 1992.

[86] T. Turletti, "The INRIA videoconferencing system (IVS)," *ConneXions - The Interoperability Report*, vol. 8, no. 10, pp. 20–24, October, 1994.

[87] M. Vogt, B. Plattner, T. Plagemann, and T. Walter, "A run-time environment for Da CaPo," in *INET*, 1993.

[88] R. W. Watson and S. A. Mamrak, "Gaining efficiency in transport services by appropriate design and implementation choices.," *ACM Transactions on Computer Systems*, vol. 5, no. 2, pp. 97–120, May, 1987.

[89] W. Wilson Ho, *DLD. A Dynamic link/Unlink Editor. Version 3.2.3.*, 1991. ftp://metro.ucc.su.oz.au/pub/gnu/dld-3.2.3.tar.Z.

[90] W. Wilson Ho and R. A. Olsson, "An approach to genuine dynamic linking," *Software – Practice and Experience*, vol. 21, no. 4, pp. 375–390, April, 1991. ftp://metro.ucc.su.oz.au/pub/gnu/dld-3.2.3.tar.Z.

[91] C. M. Woodside, K. Ravindran, and R. G. Franks, "The protocol bypass concept for high speed OSI data transfer," pp. 107–122, IFIP WG 6.1/WG 6.4, North-Holland Publ., Amsterdam, The Netherlands, November, 1990.

[92] L. Zhang, "Why TCP timers don't work well," in *SIGCOMM'86*, pp. 397–405, 1986.

[93] M. Zitterbart, B. Stiller, and A. Tantawy, "A model for flexible high-performance communication subsystems," Tech. Rep. RC17801, IBM Thomas J. Watson Research Center, February, 1992. (Appeared in the Journal on Selected Areas in Communications May 1993).