Performance of HTTP and FTP Applications over Local Area Ethernet and ATM Networks

Edwin Law

A thesis submitted to the Faculty of Graduate Studies
in partial fulfillment of the requirements
for the degree of

Master of Science

Thesis Supervisor: **Tim Brecht**

Graduate Programme in Computer Science
Department of Computer Science, York University
4700 Keele Street, North York, Ontario
M3J 1P3, Canada

Performance of HTTP and FTP Applications over Local Area Ethernet and ATM Networks

Edwin Law

A thesis submitted in conformity with the requirements
for the Degree of Master of Science
Graduate Programme in Computer Science
York University

Abstract

This thesis examines, compares, and improves the throughput of the HTTP and FTP applications when executing over the Ethernet and ATM networks. All experiments are conducted on the SGI and Sun platforms using three network environments: TCP/IP over Ethernet, TCP/IP over ATM, and native ATM using AAL5.

We have successfully ported socket-based HTTP and FTP applications to the Fore Systems ATM API, and implemented a simple window-based retransmission protocol inside these applications to ensure that the data transfers are reliable. We have shown that this reliable protocol does not introduce significant overhead to the performance of the network applications.

We have conducted extensive experiments and gained insights into the behaviour of the three network environments. When compared with the original applications, we have obtained significant increases in throughput by adjusting a number of software configurable network and protocol parameters. They include the socket buffer size, data buffer size, and ATM MTU size.

In the absence of file caching, the file systems are the major bottleneck on the performance of network applications. In our experiments, the file systems show a peak file read/write access rate of no higher than 27 Mbps on both platforms when using uncached files. This greatly restricts our network applications from fully utilizing the bandwidth provided by the ATM network.

With the applications we examine, TCP/IP over ATM and native ATM do provide increased network throughput when compared with the conventional Ethernet environment, but the improvement at the application level is far below that which the physical network medium can provide. Using the current hardware and software components available to us, TCP/IP over ATM generally yields throughputs similar to native ATM. Therefore, we believe that TCP/IP over ATM is currently sufficient in providing a means for existing networks to interoperate with ATM networks.

We conclude that, before the applications can benefit from the full potential of the ATM networks (and in fact any other high-speed network), further improvements to the hardware and software components, such as the file systems and disks, must be made. The full potential of the high-speed networks will only be realized after such improvements can be accomplished.

Acknowledgments

First, I would like to thank my supervisor, Tim Brecht, for providing me with invaluable suggestions, encouragement and support. He has always had time for discussions, and has continually shown great interest in my research.

I must also thank my committee members for consenting to be on my examination committee, and for spending their valuable time and effort.

Next, thanks must also go to all the people who have provided their assistance. Thanks to the technical staff of the Department of Computer Science, especially Norman Wilson and Matt Robinson, for patiently answering numerous questions and for providing helpful solutions to my problems. I thank Patricia Plummer, our secretary, for assisting me in bureaucratic matters. I thank Todd Derr of Fore Systems Inc. for resolving countless numbers of technical problems concerning the ATM software and hardware. I would also like to thank Lisa Salem-Wiseman for proof-reading my lengthy thesis.

Furthermore, I am thankful to all my friends and colleagues in the Department of Computer Science for creating an enjoyable and stimulating atmosphere that has made these years at the department very pleasant.

Lastly, I give my special thanks to my family for their solid support and understanding over the past three years.

Contents

Page

Abstract i

Acknowledgments ii

Table of Contents iii

List of Figures vi

List of Tables viii

Glossary ix

CHAPTER

1 Introduction 1
- 1.1 Motivation 1
- 1.2 Objectives 3
- 1.3 Contributions 3
- 1.4 Outline of the Thesis 4

2 Background 5
- 2.1 Ethernet versus ATM Local Area Networks 5
  - 2.1.1 Network Bandwidth 5
  - 2.1.2 Network Architecture 6
  - 2.1.3 Network Protocol Stacks 6
  - 2.1.4 Data Encapsulation, Segmentation and Reassembly 7
  - 2.1.5 Quality of Service 9
- 2.2 Socket Implementations 9
  - 2.2.1 BSD Implementation 10
  - 2.2.2 System V Implementation 13
  - 2.2.3 Time Traces of Network Input and Output 16
- 2.3 Fore Systems ATM Environment 19
  - 2.3.1 The ATM Hardware 19
  - 2.3.2 The ATM Software 19
  - 2.3.3 Fore System's Application Programming Interface 20
  - 2.3.4 The Fore ATM Network Implementation 20

2.4 TCP/ATM versus Native ATM networks . . . 21
2.4.1 Classical IP and Fore IP . . . 21
2.4.2 Characteristics of TCP/ATM Networks . . . 23
2.4.3 Performance of TCP/ATM and Native ATM Networks . . . 24
2.5 Network Applications . . . 26
2.5.1 Application Layer Protocols . . . 27
2.5.2 Network Applications . . . 28
2.5.3 Summary . . . 29

3 Network Applications and Porting Them to Native ATM **30**
3.1 General modification approach . . . 30
3.1.1 The Makefile and Header Files . . . 30
3.1.2 The Socket Functions . . . 30
3.1.3 Reliability and Flow Control . . . 34
3.2 Netperf . . . 36
3.2.1 Program Architecture . . . 36
3.2.2 Compilation, Setup and Execution . . . 36
3.2.3 Network Input and Output . . . 37
3.2.4 Timing . . . 37
3.2.5 Modifications for Native ATM . . . 37
3.3 TTCP . . . 37
3.3.1 Program Architecture . . . 37
3.3.2 Compilation, Setup and Execution . . . 38
3.3.3 Network Input and Output . . . 38
3.3.4 Timing . . . 38
3.3.5 Modifications for Native ATM . . . 38
3.4 NCSA HTTP Daemon . . . 38
3.4.1 Program Architecture . . . 39
3.4.2 Compilation, Setup and Execution . . . 40
3.4.3 Network Input and Output . . . 40
3.4.4 Modifications for Native ATM . . . 40
3.5 WebStone . . . 41
3.5.1 Program Architecture . . . 41
3.5.2 Compilation, Setup and Execution . . . 41
3.5.3 Network Input and Output . . . 42
3.5.4 Timing . . . 42
3.5.5 Modifications for Native ATM . . . 43
3.6 Washington University FTP Daemon . . . 43
3.6.1 Program Architecture . . . 43
3.6.2 Compilation, Setup and Execution . . . 44
3.6.3 Network Input and Output . . . 44
3.6.4 Modifications for Native ATM . . . 44
3.7 FreeBSD FTP . . . 45
3.7.1 Program Architecture . . . 45
3.7.2 Compilation, Setup and Execution . . . 45
3.7.3 Network Input and Output . . . 45
3.7.4 Timing . . . 46

3.7.5 Modifications for Native ATM 46
3.8 Summary of Modifications 47

4 Experiments **48**
4.1 The York University LCSR Testbed 48
4.1.1 Hardware and Software Configuration 48
4.1.2 Network Configuration 49
4.2 Outline of the Experiments 49
4.3 Memory-to-Memory Transfer 53
4.3.1 Methodology 53
4.3.2 Ethernet 53
4.3.3 TCP/ATM 58
4.3.4 Native ATM 62
4.3.5 Comparison of Netperf and TTCP Results 65
4.3.6 Reliable Native ATM 65
4.3.7 Summary of Memory-to-Memory Transfers 68
4.4 File Access 70
4.4.1 Methodology 70
4.4.2 File Read Access 72
4.4.3 File Write Access 75
4.4.4 Summary of File Access Experiments 76
4.5 HTTP Transfers 78
4.5.1 Methodology 78
4.5.2 Ethernet 80
4.5.3 TCP/ATM 81
4.5.4 Native ATM 82
4.5.5 Summary of HTTP Transfers 85
4.6 FTP Transfer 87
4.6.1 Methodology 87
4.6.2 Ethernet 88
4.6.3 TCP/ATM 89
4.6.4 Native ATM 91
4.6.5 Summary of FTP Transfers 96
4.7 Summary 98

5 Conclusions and Future Work **101**
5.1 Contributions 101
5.2 Future Work 104

Bibliography **106**

List of Figures

2.1 Ethernet and ATM Network architectures 6
2.2 Ethernet, IP and TCP encapsulations 8
2.3 AAL5 cell preparation 9
2.4 Example of an mbuf chain under IRIX 6.2 11
2.5 Network output under BSD Sockets 11
2.6 Network input under BSD Sockets 12
2.7 STREAMS implementation of TCP/IP over Ethernet 14
2.8 Sockets implementation under SVR4 15
2.9 Time traces of Network input and output (SGI's) 17
2.10 Time traces of Network input and output (Sun's) 18
2.11 Fore System's software/hardware environment [24] 20
2.12 LLC/SNAP encapsulation of IP packets 22
2.13 Fore IP encapsulation of IP packets 22
2.14 A typical FTP client/server scenario 28

3.1 Socket API vs Fore API server implementations 31
3.2 Socket API vs Fore API client implementations 32
3.3 The reliable flow-control algorithm 35
3.4 Example of the reliable flow-control algorithm 36
3.5 Pre-forking in NCSA HTTPd 39
3.6 WebStone architecture 41
3.7 Wu-ftpd via the Internet Daemon 43

4.1 The LCSR testbed 49
4.2 Measuring data transfer time using different benchmarks 50
4.3 Netperf memory-to-memory transfer (Ethernet, SGI) — socket buffer test 55
4.4 Netperf memory-to-memory transfer (Ethernet, SGI) — peak throughput 55
4.5 Netperf memory-to-memory transfer (Ethernet, SGI) – data buffer test 56
4.6 Netperf memory-to-memory transfer (Ethernet, Sun) – socket buffer test 57
4.7 Netperf memory-to-memory transfer (Ethernet, Sun) – data buffer test 57
4.8 Netperf memory-to-memory transfer (TCP/ATM, SGI) — socket buffer test 59
4.9 Adjustment of socket buffer size under BSD Sockets 59
4.10 Netperf memory-to-memory transfer (TCP/ATM, SGI) — data buffer test 60
4.11 Netperf memory-to-memory transfer (TCP/ATM, Sun) — socket buffer test 61
4.12 Netperf memory-to-memory transfer (TCP/ATM, Sun) — data buffer test 61
4.13 Netperf memory-to-memory transfer over unreliable native ATM (SGI) 64
4.14 Netperf memory-to-memory transfer over unreliable native ATM (Sun) 64
4.15 Comparison of Netperf and TTCP results 66

4.16 TTCP memory-to-memory transfer (SGI, reliable native ATM) 67
4.17 TTCP memory-to-memory transfer (Sun, reliable native ATM) 68
4.18 File read access rates (SGI, uncached files) 72
4.19 File read access rates (SGI, cached files) 73
4.20 File read access rates (Sun, uncached files) 74
4.21 File read access rates (Sun, cached files) 74
4.22 File write access rates (SGI, uncached files) 75
4.23 File write access rates (SGI, cached files) 76
4.24 File write access rates (Sun, uncached files) 77
4.25 File write access rates (Sun, cached files) 77
4.26 HTTP transfers over Ethernet 81
4.27 HTTP transfers over TCP/ATM 82
4.28 HTTP transfers over unreliable native ATM 83
4.29 Webstone file transfer over reliable native ATM 84
4.30 Summary of HTTP transfers 85
4.31 FTP transfers over Ethernet (SGI) 89
4.32 FTP transfers over Ethernet (Sun) 90
4.33 FTP transfers over TCP/ATM (SGI) 91
4.34 FTP transfers over TCP/ATM (Sun) 92
4.35 FTP transfers over unreliable native ATM (SGI) 93
4.36 FTP transfers over reliable native ATM (SGI) 93
4.37 FTP transfers over unreliable native ATM (Sun) 95
4.38 FTP transfers over reliable native ATM (Sun) 95
4.39 Summary of FTP transfers 97
4.40 Summary of experimental results (SGI) 99
4.41 Summary of experimental results (Sun) 100

List of Tables

2.1 Message block allocations 14
2.2 Maximum TCP/ATM throughputs from various studies 24
2.3 Maximum native ATM throughputs from various studies 25

3.1 Summary of default data buffer sizes 47
3.2 Summary of modifications of the network benchmarks/applications 47

4.1 The memory-to-memory transfer experiments 54
4.2 Memory-to-memory transfer improvements using the optimal buffer sizes 70
4.3 The file access experiments 71
4.4 Maximum file access rates 77
4.5 Network parameters used in the HTTP transfer experiments 79
4.6 The Webstone standard file set [55] 79
4.7 Data loss in HTTP transfers over unreliable native ATM 83
4.8 Number of retransmissions in HTTP transfers over reliable native ATM 85
4.9 HTTP transfer improvements using the optimal buffer sizes (5 MB file) 86
4.10 Network parameters used in the FTP transfer experiments 88
4.11 Data loss in FTP transfers over unreliable native ATM 92
4.12 Number of retransmissions in FTP transfers over reliable native ATM 94
4.13 FTP transfer improvements using the optimal buffer sizes (50 MB file) 98

Glossary

AAL (ATM Adaptation Layer) — An ATM protocol layer that converts the variable-length data of the application layer to the 53-byte cell format of the ATM layer.

API (Application Program Interface) — The interface by which an application program accesses operating system and other services.

Application Layer — The top layer of the OSI model which defines the protocols used between application programs.

ASAP (Application Service Access Point) — A unique address used in the Fore API, which is assigned to every connection end-point within the same ATM device.

ATM (Asynchronous Transfer Mode) — A network technology based on asynchronous time division multiplexing using fixed-length data units.

ATM End-point — A connection end-point used by a native ATM application. The address of an ATM end-point is comprised of an NSAP and an ASAP.

ATM Layer — An ATM protocol layer that handles most of the processing and routing activities such as building the ATM header, cell multiplexing/cell demultiplexing, and flow control.

ATM MTU — The maximum size of the CPCS-PDU allowed in an ATM network.

ATM Physical Layer — The bottom layer of the ATM protocol stack which defines the interface between ATM traffic and the physical media.

ATM Window Size — The number of bytes a sender transmits before it requires an acknowledgment from the receiver in our simple reliable native ATM network protocol.

BSD (Berkeley Software Distribution) — A family of Unix versions developed at the University of California at Berkeley.

CPCS-PDU (Common Part Convergence Sublayer Protocol Data Unit) — A data unit that contains information that conforms to the specifications of the ATM convergence sublayer, and is ready to be segmented into ATM cells.

CS (Convergence Sub-layer) — The portion of the AAL that prepares information in a common format (CPCS-PDU) before it is segmented into ATM cells and returns it to its original form after reassembly at the destination.

Data Buffer Size — The size of memory buffer used by an application to transmit or receive data. A pointer to this buffer is usually passed as an argument to the responsible network input/output function call.

Data Link Layer — The second lowest layer of the OSI model which defines how the network layer packets are transmitted as bits.

Data Size — The amount of data transferred in a network application (e.g., HTTP transfer).

Ethernet — A common standard for connecting computers into a local area network. In this thesis, "Ethernet" is used to refer to the traditional Ethernet network with a bandwidth of 10 Mbps, and also to refer to the TCP/IP over Ethernet network environment.

Fore IP — A implementation developed by Fore Systems to enable the IP layer to operate on top of the ATM network.

FTP (File Transfer Protocol) — An application level protocol for transferring files over a TCP/IP network.

Gbps (Gigabits per second) — A way of measuring network usage or bandwidth. In this thesis, "giga" means $1024 \times 1024 \times 1024$.

HTTP (Hyper Text Transfer Protocol) — An application level protocol used on the World-Wide-Web for the exchange of HTML documents.

IEEE (Institute of Electrical and Electronics Engineers) — A publishing and standards-making body for the electronics industry.

IETF (Internet Engineering Task Force) — An organization that provides coordination of standards and specifications development for TCP/IP networking.

Internet Daemon (inetd) — A server process that listens for service requests on several TCP or UDP ports. When a request arrives, it executes the server program associated with the service.

IP (Internet Protocol) — The network layer protocol for the TCP/IP protocol suite. It is a connectionless, best-effort, packet switching protocol.

IP/ATM (IP over ATM) — A network environment in which the IP layer operates on top of the ATM network.

ISO (International Standards Organization) — A voluntary organization responsible for creating international standards in many areas, including computers and communications.

LAN (Local Area Network) — A data communications network which is limited to the immediate area, usually the same building or floor of a building.

Mbps (Megabits per second) — A way of measuring network usage or bandwidth. In this thesis, "mega" means 1024×1024.

Mbuf — Fixed size memory buffers residing in the kernel address space which can be shared by various protocol stacks in BSD systems.

MSS (Maximum Segment Size) — The maximum amount of TCP data that a node can send in one TCP segment.

MTU (Maximum Transmission Unit) — The largest data unit which may be sent on a physical medium.

Native ATM — A network environment which allows features of the underlying ATM network such as permanent virtual circuits and end-to-end quality of service to be accessible by user applications.

Network Layer — The third lowest layer of the OSI model which defines how information from the transport layer is sent over networks and how different hosts are addressed.

NIC (Network Interface Card) — An adapter card installed in a computer to provide a physical connection to a network.

NSAP (Network Service Access Point) — In this thesis, this refers to the unique address which is assigned to every ATM device.

OSI (Open Systems Interconnection) — A model of a network architecture and a suite of protocols (a protocol stack) developed by the ISO as a framework for international standards in heterogeneous computer network architecture.

Physical Layer — The lowest layer of the OSI model which defines how bits are converted into electrical current, light pulses or any other physical form.

QoS (Quality of Service) — A set of quality requirements that characterize the traffic over a given virtual connection.

SAR (Segmentation and Reassembly Sublayer) — The portion of the AAL that segments the CPCS-PDU into 48-byte payloads and reassembles 48-byte payloads into the CPCS-PDU.

Socket — A mechanism for creating a virtual connection between processes. A socket is used for interprocess communications as well as network input/output operations in typical UNIX systems. A socket has associated with it a socket address, consisting of a port number and the local host's network address.

Socket Buffer size — The size of a sender or receiver buffer used by a socket under the BSD sockets implementation.

STREAMS — An I/O subsystem that provides full duplex paths between user processes and hardware devices (or pseudo-devices).

TCP (Transmission Control Protocol) — A transport layer protocol for the TCP/IP protocol suite which adds reliable communication, flow-control, multiplexing and connection-oriented communication to the underlying network layer protocol.

TCP/ATM (TCP over ATM) — A network environment in which the TCP and IP layers operate on top of an ATM network.

TCP segment — A protocol data unit that consists of TCP header information and optional data.

Throughput — The amount of data transmitted between two points in a given amount of time. All throughput measurements in this thesis are reported in megabits per second (Mbps), which is equivalent to 1024×1024 bits per second.

Transport Layer — The fourth lowest layer of the OSI model which takes care of data transfer, ensuring the integrity of data if desired by the upper layers.

UDP (User Datagram Protocol) — A transport layer protocol for the TCP/IP protocol suite, which provides simple, connectionless, and unreliable datagram services.

WAN (Wide Area Network) — A data communications network that covers an area larger than a single building or campus.

Chapter 1

Introduction

Since the early 1970's, computers have been interconnected by networks such as Ethernet and Token Ring. As a result of the continuing dramatic increase in the volume of data being transferred and of the demand for higher communication bandwidth, these types of networks are no longer adequate to handle the explosion of Internet traffic. Although new network technologies provide us with higher network bandwidths, there is still a lack of studies examining performance issues of network applications executing over these networks.

In this thesis, we examine and improve the throughput of network applications when executing over Ethernet and ATM networks. Numerous experiments are conducted to compare the throughput of these applications when using three different network environments: TCP/IP over Ethernet (which forms the basis for our comparisons), TCP/IP over ATM, and native ATM using AAL5 along with a simple reliable protocol we have built on top of it. The experimental results are analyzed in order to gain a better understanding of the behaviour of the network applications when executing using the three network environments, and in order to understand how user and application level modifications to the software, and to various software configurable network and protocol parameters, can affect the throughput of these applications.

1.1 Motivation

Asynchronous Transfer Mode (ATM) [34, 37] technology marks the beginning of a new era of network integration and performance. It has been universally accepted as the transfer mode of choice for Broadband Integrated Services Digital Network (BISDN), and is designed for the high-speed transfer of voice, video, and data through computer networks. ATM technology is still evolving, and as a result, so are its standards. Apart from its extraordinary speed, its other advantages over conventional networking technologies include scalability, as well as flexible and guaranteed bandwidth allocation. This revolutionary network technology has already gained rapid acceptance in the local area networking community.

As promising as the ATM technology is, the ability to maintain backward compatibility with existing protocols and software is critical. Unfortunately, most of today's network software and applications are not interoperable with native ATM protocols. The cost of upgrading legacy software and hardware could be enormous. With the presence of competition from other emerging network technologies, the migration of existing networks to ATM networks may not be warranted, unless ATM network technologies can be shown to have significant advantages over the others.

The advantages that the ATM technology provides (namely its scalability, flexible and guaranteed bandwidth allocation) are generally accepted by the networking community. However, there are different ways to make use of these advantages. They include the following:

- Run network applications directly on top of the ATM network using ATM protocol layers. These are commonly known as "native ATM" applications. However, the major drawback of this approach is that the API's provided for native ATM programs are generally incompatible with existing network applications (the ATM protocol stack does not fit into the conventional OSI model [18]). Hence, modifications to the current applications are required. These applications may also need to support essential functions not provided by the ATM protocol layers (such as reliability).

- Emulate legacy network protocols on top of the ATM networks. The ATM Forum's LAN Emulation (LANE) [6] and Multiprotocol Over ATM (MPOA) [7] are two example standards. IETF has developed another standard known as Classical IP over ATM [38]. Emulating legacy network protocols on top of ATM networks is definitely less costly in comparison with a complete software and hardware migration to the ATM networks. However, this approach raises incompatibility issues, such as IP address resolution over ATM networks. The additional protocol layers will introduce redundancies and overheads.

- Reject ATM technology as a whole but provide some of the ATM functionalities as an extension to existing network protocols. For example, the Internet Engineering Task Force (IETF) has defined quality of service (QoS) protocols on top of IP networks, such as Resource Reservation Protocol (RSVP) [11] and Real-Time Transport Protocol (RTP) [53]. However, due to the completely different nature of the ATM network, it cannot possibly be emulated completely on top of conventional networks.

There is currently no agreement within the networking community as to which of the three approaches is the best to pursue. It is essential to further investigate these approaches so that a clear direction can be established for the future development of computer networking applications.

Previous studies show that native ATM network environments out-perform Ethernet networks in throughput measurements [31, 41], and that TCP/IP over ATM (also known as TCP/ATM), although performing considerably faster than Ethernet [40, 26, 41, 1], introduces significant overheads which prevent the networks from achieving their maximum bandwidths [4, 13, 40]. However, these studies only measure the throughput and round-trip times of raw data transfers using simple network benchmarks, and almost all of them are performed on data transfers from one machine's memory to that of another. Studies done on real applications using TCP/ATM are limited, and those using native ATM applications are rarely seen (we have only encountered two other preliminary investigations [8, 19] of the issue prior to our work). There is no evidence to show that network performance of raw data transfers can closely reflect the performance of actual network applications, since the applications may introduce other latencies and bottlenecks. It is therefore necessary to examine the performance of real network applications.

Many performance studies of network applications over traditional networks (e.g., a comparison study of several HTTP daemon programs by McGrath [43]) are conducted without considering the effect of network and protocol parameters. Furthermore, a few previous studies [17, 45, 40] point out that when using the TCP/IP over ATM environment, poor choice of network parameters may lead to unexpectedly low throughput, even lower than that obtained with the Ethernet network (Section 2.4.2 describes this phenomenon in more detail). In this thesis, one of our goals is to discover how these parameters can be adjusted and how they impact the throughput of the applications considered.

There are a variety of application level protocols implemented by today's network-oriented applications. Currently, the HTTP and FTP protocols are used predominantly by the Internet community. Indeed, many recent studies [27, 46, 61] show that HTTP and FTP applications are the two most frequently used network applications. We therefore concentrate on these two applications in our investigation.

1.2 Objectives

The main goal of this thesis is to gain a better understanding of the performance of network applications which execute using TCP/IP over Ethernet, TCP/IP over ATM, and native ATM (using AAL5) network environments. Additionally, much of the work in this thesis is driven by the following objectives:

- We want to observe the performance of network applications executing over the TCP/IP over ATM and native ATM network environments, using the 10 Mbps Ethernet network as the base.
- Although TCP/IP over ATM provides interoperability between the legacy IP-based software and ATM networks, we want to see if the amount of overhead introduced in order to provide this compatibility significantly limits the throughput of such applications.
- We want to investigate the feasibility of running network applications using native ATM network environments. This may avoid potential overheads and redundancies, but other issues must be addressed (e.g., adding reliability to the native ATM applications).
- We want to determine how various software configurable parameters can impact the throughput of network applications. In particular, we consider the application's data buffer size, the socket buffer size, and the MTU size used by the ATM network (ATM MTU).

1.3 Contributions

In this thesis, we examine, compare, and improve the throughput of two of the most widely used Internet applications (HTTP and FTP) when executing using the TCP/IP over Ethernet, TCP/IP over ATM, and native ATM local area network environments.

In order to measure the throughput obtained by real network applications when executing using a native ATM network environment, we modify the HTTP and FTP applications, which were originally written using the Sockets Application Programming Interface, to a programming interface designed for ATM networks and supplied by Fore Systems Incorporated with their network interface cards. Currently, the ATM protocols do not guarantee reliable data transfer between hosts. Therefore, we implement a simple window-based retransmission protocol inside the native ATM applications to ensure that data is transfered reliably.

We conduct numerous experiments on two different platforms (the Sun Sparcstation's and SGI Indy's), in order to reduce the likelihood of any biases in conclusions drawn based on only one particular platform.

Our investigation begins by measuring the throughput of memory-to-memory transfers (i.e., raw data transfers from the memory of an application on one host to that of another host) in the three network environments. This is done in order to reveal their characteristics, such as the peak throughput and the effects of various software configurable parameters on the throughput. We then use these results to tune our HTTP and FTP applications for the best performance possible. The parameters studied include the socket buffer size, the ATM MTU size, and the application's data buffer size. We show that by suitably adjusting these parameters from their default values, significant improvements in throughput are obtained. For example, we are able to increase the peak throughput using the Ethernet environment on the SGI's from 6.5 Mbps to 8.1 Mbps by changing the socket buffer size from its default value. Similar improvements are obtained by modifying the HTTP and FTP applications used.

We also investigate the throughput that is obtained when reading and writing files (we call this the "file access rate"). Using uncached files, we find that relatively high access rates (between 24 Mbps and 27 Mbps) are obtained when using large files (5MB and 50MB), but low access rates (less than 2 Mbps for read access) are obtained when using small files (500 bytes and 5KB).

The HTTP and FTP experiments show that the throughput of the network applications when using the Ethernet are mainly limited by the network bandwidth of 10 Mbps, whereas the throughput obtained by those applications using TCP/ATM and native ATM network environments are relatively far from the theoretical maximum. We show that in these two environments, throughput is greatly influenced by the choice of socket and data buffer sizes, as well as ATM MTU size. We found the latter to be the main factor in the improvements obtained in the TCP/ATM environment. We also find that in these two environments, the file access rates greatly limit the throughput of FTP applications which use uncached files.

Our experiments conducted using the Netperf and TTCP benchmarks show that the throughput of memory-to-memory transfers is significantly higher than the throughput observed when running real network applications. However, the performance results obtained by these simple benchmarks are useful in determining an upper bound for the network applications. In this respect we see that the application level performance is still far below that which the physical network medium can provide. Before deploying any high speed network that fully exploits its available bandwidth, significant improvements are needed in the areas of disks and file system designs, as well as file caching techniques.

1.4 Outline of the Thesis

The organization of this thesis is as follows: in Chapter 2, we review background related to conventional TCP/IP over Ethernet and ATM networks and provide a brief overview of the HTTP and FTP protocols. Chapter 3 describes in detail the benchmarks and network applications used in our experiments, as well as the modifications required to port them to the native ATM environment. Chapter 4 describes our experiments and the results. Lastly, Chapter 5 presents the conclusions of this thesis and suggestions for future work.

Chapter 2

Background

In this chapter we present background and research related to this thesis. We begin by discussing the fundamental differences between Ethernet and ATM networks. We then describe the implementation of sockets at the operating system level. An overview of the ATM hardware and software follows. We also look at two approaches for executing applications over ATM networks: TCP/IP over ATM (also known as TCP over ATM, or TCP/ATM) and native ATM (i.e., no intermediate layer between the ATM and application layers). We then present work related to the performance of ATM networks. Lastly, we introduce the FTP and HTTP protocols and the network applications used in our experiments.

2.1 Ethernet versus ATM Local Area Networks

Ethernet [57] is a local area network (LAN) technology which is commonly used today. The Ethernet standard (also called the DIX standard) [20] was published in 1982 by Xerox research. The Institute of Electrical and Electronics Engineers (IEEE) later developed another standard for Ethernet [32] (also called the 802.3 standard). Asynchronous Transfer Mode (ATM) networks [34, 37] use comparatively new technologies designed for Broadband Integrated Services Digital Network (B-ISDN) [33] to transfer voice, video and data through computer networks at very high speeds.

This thesis investigates application performance when run over Ethernet and ATM networks. Therefore, we begin with an overview of the characteristics of Ethernet and ATM networks, and the major differences between them. They include the network bandwidth, network architecture, protocol stack, data encapsulation, and quality of service.

2.1.1 Network Bandwidth

In a comparison of the Ethernet and ATM technologies, it would be insufficient to simply consider their bandwidths, since bandwidths have increased tremendously over recent years.

The original Ethernet standard, defined in the 1980's, allows information to be transmitted between computers at 10 megabits per second (Mbps). The Fast Ethernet standard introduced in 1994 enables networks to be run at transfer rates of 100 Mbps. The Gigabit Ethernet [60] (standard was finalized in June 1998) offers a 1 gigabit per second (Gbps) raw bandwidth (i.e., 100 times faster than the original Ethernet).

On the other hand, most ATM networks today run at transfer rates of 100 Mbps, 155 Mbps or 622 Mbps. Higher speeds for ATM networks, including 1.244 Gbps, 2.488 Gbps, and even higher, will become available over the next few years.

All these technologies are beneficial to us only if their network bandwidths can be efficiently utilized by the network applications. Thus, it is important to conduct performance studies (such as throughput measurements) of the network applications executing over these networks.

2.1.2 Network Architecture

Figure 2.1 shows the architectures of typical Ethernet and ATM networks. An Ethernet network consists of machines (also known as stations) attached to a shared medium. Access to the shared medium is determined by the Medium Access Control (MAC) mechanism embedded in each station's Network Interface Card (NIC). An Ethernet packet can only be sent when the medium is idle. If two stations happen to transmit at the same instant, their signals collide and the transmissions are rescheduled. With the existence of a shared medium, as more stations are added and more network applications are deployed, data traffic will increase to such levels that the network becomes overloaded, and performance may degrade significantly due to collisions.

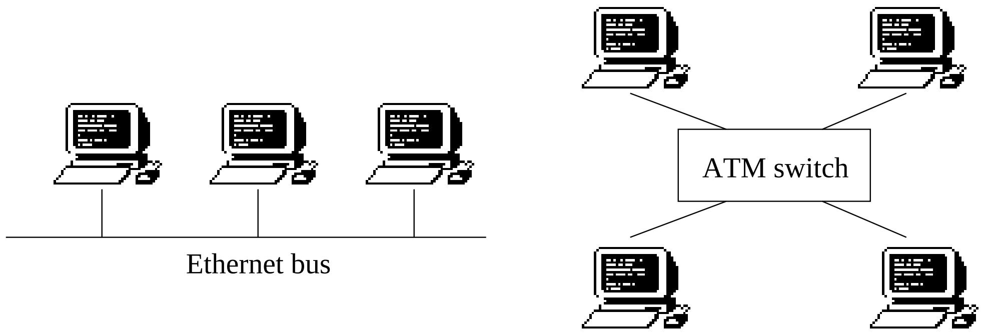


Figure 2.1: Ethernet and ATM Network architectures

The basic component of an ATM network is a special purpose electronic switch designed to transfer data at extremely high speeds. The ATM switch is connected to the NIC's of the stations, and relays data from one station to another. In an ATM network, the only shared resource is the switch, which is capable of supporting multiple and simultaneous data transfers without severe bandwidth degradation.

"Switched Ethernet" [2] (also known as "Ethernet Switching") is another emerging technology that attempts to add the advantages of switch-based networks to traditional Ethernet networks. This is achieved by introducing switches to the network to reduce medium sharing. This approach is generally less costly than an ATM network, although it still lacks other beneficial properties that are inherent in the ATM network (e.g., quality of service, see Section 2.1.5). In fact, Ethernet switching is considered by some to be an intermediate step in migrating from Ethernet to ATM networks [2]. This also strongly indicates that future network technologies are likely to be switch-based.

The equipment provided in our testbed is insufficient for us to perform congestion analysis of network applications executing over the two types of networks, and therefore we do not consider the impact of congestion in this thesis. Hence, the impact which the architectural differences between the network environments has on the throughput may not be obvious in our experimental results.

2.1.3 Network Protocol Stacks

Conventional network protocols typically follow the International Standards Organization (ISO) reference model for the Open Systems Interconnection standard (OSI) [18]. Ethernet fits into the physical and data link layers (layers 1 and 2) of the model. On top of Ethernet, the Internet Protocol (IP) [49] is typically used. IP is a network layer protocol (OSI layer 3) which provides a connectionless and unreliable delivery system. It forwards data based on a uniquely assigned 32-bit destination address (IP address). Two transport layer protocols (OSI layer 4) are commonly used directly above IP. The User Datagram Protocol (UDP) [48] provides a connectionless service with no guarantee that the data ever reaches its destination, whereas the Transmission Control Protocol (TCP) [50] provides a connection-oriented and reliable service. The protocol stack we investigate uses TCP and IP over a conventional 10 Mbps Ethernet network used in our testbed (abbreviated as "TCP/IP/Ethernet", or simply "Ethernet").

There are three distinct layers in the ATM protocol stack: the physical layer, ATM layer, and ATM adaptation layer (AAL). At the lowest level, the physical layer controls how bits are transmitted on the physical medium, keeps track of ATM cell boundaries, and packages cells into the appropriate type of frame for the physical medium being used. The ATM layer is responsible for establishing connections and passing cells through the ATM network. The AAL receives packets from upper-level protocols and breaks them into the 48-byte segments that form the payload of an ATM cell. There are currently four types of AAL's defined: AAL1 supports connection-oriented services that require constant bit rates and have specific timing and delay requirements; AAL2 supports connection-oriented services that do not require constant bit rates; AAL3/4 is intended for both connectionless and connection-oriented variable bit rate services; and AAL5 supports connection-oriented variable bit rate data services.

Because the ATM network switches cells over the hardware layer from router to router, it might seem that its protocol stack fits into the data link layer. However, it also performs connection negotiation, flow control and routing, none of which are part of the data link layer. In fact, the ATM protocol stack does not exactly fit into the OSI model.

The TCP/ATM network environment (see Section 2.4) used in our experiments provides mechanisms to interoperate IP with the ATM layers. In other words, it attempts to place the TCP/IP protocol stack on top of the AAL/ATM protocol stack. This redundancy of protocol layers is one of the major shortcomings of this network environment.

2.1.4 Data Encapsulation, Segmentation and Reassembly

Ethernet and ATM networks use different schemes to segment and encapsulate data into packets for transportation and reassembly at the destination. As a result, different amounts of protocol overhead are introduced by the two schemes. We believe these differences can potentially influence the throughput of our network applications.

TCP prepends a 20-byte header to the data from an application and packages it into an "TCP segment". The maximum size of data that can be stored in a TCP segment is known as the Maximum Segment Size (MSS). This size can vary from one machine to another, and is negotiated when the connection is set up. The IP layer prepends a 20-byte header to the data passed from the higher layer (such as TCP) to become an "IP datagram". Fragmentation is performed if this datagram is larger than the Maximum Transmission Unit (MTU), a limit imposed by the data link protocol. The Ethernet used in our testbed imposes a limit on the MTU of 1500 bytes including the TCP and IP headers, leaving 1460 bytes for data. An IP datagram that is small enough to avoid fragmentation is called an "IP packet" [57]. The TCP MSS is always bounded by the MTU size and therefore IP layer fragmentation is not required for TCP over IP. A 14-byte header is added to the front and a 4-byte checksum to the end of the IP packet to make up a 1518 byte Ethernet frame, which is then passed down to the network interface for transmission. Data is received by following the reverse path. Figure 2.2 shows an encapsulation of one TCP segment using a 1460-byte MSS and a 1500-byte MTU. Section 2.2 describes in detail our machines' segmentation and reassembly implementations using the TCP/IP over Ethernet environment.

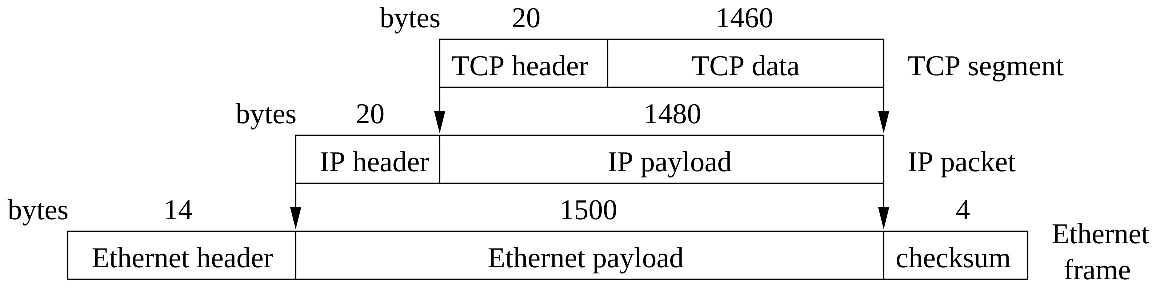


Figure 2.2: Ethernet, IP and TCP encapsulations

Unlike Ethernet or IP packets, ATM cells are fixed in size. Five of the 53 bytes of each ATM cell are reserved for the cell header, leaving 48 bytes for the data ("payload"). The use of fixed-length cells allows them to be handled faster and easier by the hardware, and enables network delays to be somewhat predictable, making ATM more suitable for carrying real-time information (for example, voice and video) as well as data.

The hardware in our testbed currently supports AAL3/4 and AAL5 only. Like the IP protocol, AAL5 performs a checksum on the whole data unit, known as the "common part convergence sublayer protocol data unit" (CPCS-PDU). On the other hand, AAL3/4 provides a checksum on each ATM cell. Another fundamental difference between the two AAL's is the amount of data per cell — 48 bytes for AAL5 and 44 bytes for AAL3/4 (4 bytes are used for the checksum). Since AAL5 was developed to provide a more efficient AAL for data communications than AAL3/4 [47], it is selected for use in our study.

We now describe AAL5, which is used in our experiments. As shown in Figure 2.3, data from the application layer that is passed down to AAL5 must be smaller than 65,536 bytes (64 KB) [30]. The convergence sublayer (CS) of AAL5 appends a variable-length padding and an 8-byte trailer to the data to become a CPCS-PDU. The padding ensures that the resulting CPCS-PDU falls on the 48-byte boundary of an ATM cell. The segmentation and reassembly (SAR) sublayer then segments the CPCS-PDU into 48-byte blocks and the ATM layer places each block into the payload field of an ATM cell, setting the "Payload Type" (PT) field of the cell header to "1" for the last cell and "0" for the rest. When the cell arrives at its destination, the ATM layer extracts the payload field from the cell. The SAR sublayer reassembles the CPCS-PDU from the SAR-PDU's. The last SAR-PDU segment of the CPCS-PDU is recognized by a value of "1" in the cell header's Payload Type (PT) field. Finally, the CS examines the CPCS-PDU's checksum in the 8-byte trailer to verify that the data has been transmitted and reassembled correctly, and then passes it up to the above layer.

Existing ATM protocols leave the responsibility of reliability to the application layer. Our native ATM applications (those running directly on top of the AAL5 layer) therefore need to include some kind of reliability scheme. Although a few retransmission schemes have been proposed in the literature (e.g., [14]), we implement our own scheme, which is simpler and easier to handle when porting the network applications to native ATM.

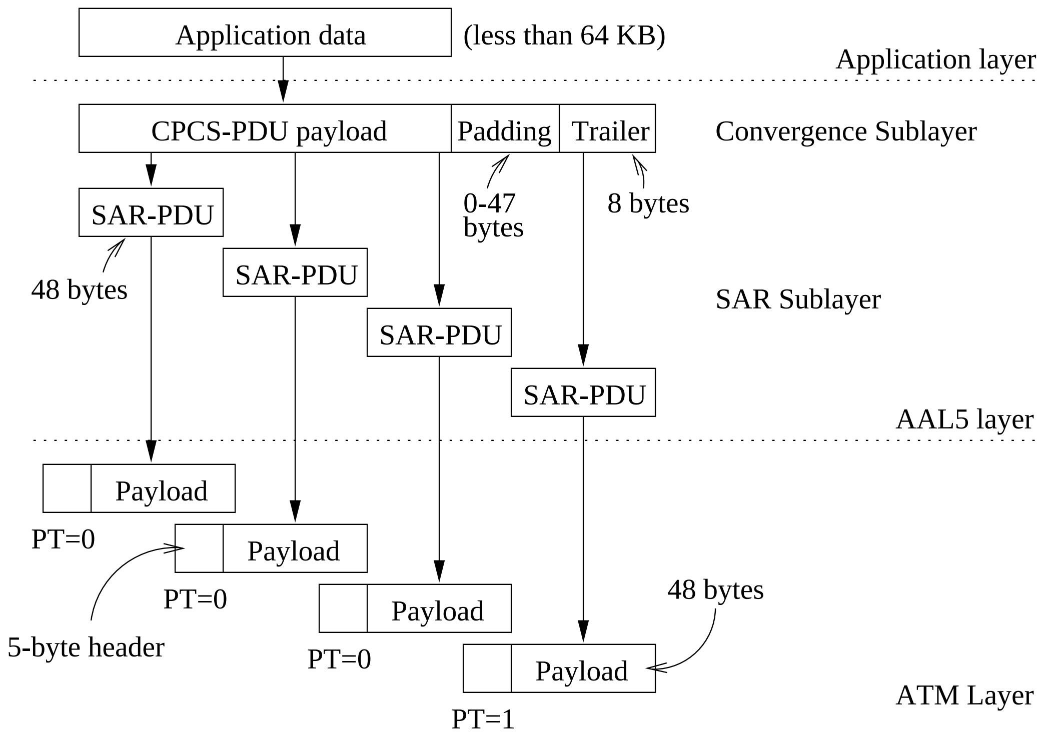


Figure 2.3: AAL5 cell preparation

2.1.5 Quality of Service

The efficiency of a particular network can be affected by the way resources are allocated. In an Ethernet network, bandwidth is allocated to an application according to the network resources available at that time. Thus, the network throughput of an application would drop substantially in a congested network. ATM instead offers statistical guarantees of the quality of service (QoS), allowing network resources to be flexibly utilized and bandwidths to be reserved and guaranteed for time-critical applications.

The quality of service component is often considered to be one of the main advantages of ATM networks in comparison to conventional network technologies. The Resource Reservation Protocol (RSVP) [11], designed for IP-based networks, is considered to be a strong competitor. However, one main disadvantage of RSVP is that it is designed only as a "best-effort" QoS — although a request for a particular QoS will be acknowledged, the network may or may not actually deliver it.

ATM QoS allows the service class to be specified as Constant Bit Rate (CBR), Variable Bit Rate (VBR, which itself is divided into real-time and non-real-time), Available Bit Rate (ABR, in which a fair portion is allocated from the available bandwidth), or Unspecified Bit Rate (UBR, in which no guarantees are made). The Fore API (see Section 2.3.3) used in our experiments is capable of supporting CBR, VBR and UBR. We have chosen to use UBR because we believe it is most likely the service class that would be utilized in a real LAN environment.

2.2 Socket Implementations

This section provides an overview of the implementation of sockets at the operating system's level. We believe that the way in which data is handled inside the operating system can affect the throughput of network applications.

In their study, Clark *et al.* [16] identify the operating system and memory copying costs as being two major sources of overheads in TCP/IP networks (the third cause is the TCP protocol). Several expensive operations are carried out by the operating system. Operations such as interrupt handling, buffer allocation/deallocation, and I/O initialization require a significant amount of processing time. The time required to move data around also adds to overhead. During each network I/O operation, data is copied to and from the user and kernel address spaces, and in some cases within the kernel. The memory cycles required for each copy, the size of the data copied, and memory cycle time all limit the speed of the data transfer.

The two operating systems used in our study are IRIX 6.2 on the SGI Indy's and Solaris 2.5.1 (SunOS 5.5.1) on the Sun Sparcstation5's. They implement networking differently and are therefore discussed separately in this section.

2.2.1 BSD Implementation

By default, IRIX 6.2 uses a socket implementation that has been derived from the BSD implementation [54]. Because the operating system's source code is not available, we describe the 4.4BSD implementation, since IRIX 6.2 uses a similar approach. It should be mentioned that IRIX also provides a STREAMS environment (see Section 2.2.2) for other protocols, to allow non-TCP/IP protocols to perform network I/O on the LAN interfaces (e.g., Fore API described in Section 2.3.3).

In the 4.4BSD operating system [44], networking memory-management facilities utilize a data structure known as an "mbuf", which is a fixed-size memory buffer residing in the kernel address space, and which can be shared by various protocol stacks. The number of mbufs available to the system is dynamically configured by the kernel, based on the amount of physical memory in the system [44, 54]. The mbuf structure consists of header and data portions. To transmit data larger than the size of the data portion, several mbufs can be linked together to form an "mbuf chain" for storing data. The first mbuf in the chain contains a "packet header" which stores additional information about the chain. Under IRIX 6.2, an mbuf is 128 bytes — 16 bytes make up the mbuf header and the remaining 112 bytes are used for data. For an mbuf chain, the packet header requires 8 bytes of storage, leaving the remaining 104 bytes for data. Instead of storing data inside an mbuf, the mbuf can reference a fixed size memory block called an external "mbuf cluster". This often reduces the number of mbufs, and since mbuf clusters are copied by reference (only pointers to the memory blocks are copied), this approach can improve efficiency tremendously. The maximum size of an mbuf cluster in IRIX 6.2 is 2048 bytes (2 KB) by default. Figure 2.4 shows three mbufs in a chain, with the second mbuf referencing an external cluster. Data is stored either in the internal data area or in the external cluster, but never in both [44, 59].

A socket maintains two structures known as the "send buffer" and "receive buffers"; each contains a queue of mbufs or mbuf chains. Each buffer has an upper limit known as the "socket buffer size" to bound the size of the queue. This upper limit refers to the total size of the mbuf chain, including mbuf headers and additional mbuf information [59]. Under IRIX 6.2, the default size is 61,440 bytes (60 KB), while the maximum size allowed is 262,144 bytes (256 KB). Whenever an application creates a socket, the two socket buffers are set up and their sizes can be changed using the `setsockopt()` call. The changing of these sizes is local to the application and therefore does not affect other applications.

Figure 2.5 shows how data is sent using 4.4BSD sockets. For example, to send 10 KB of data to the network, the application issues a system call such as `write()`, `writev()`, `sendto()` or `sendmsg()`, passing a pointer to a buffer containing the data, and also its size which we call "data

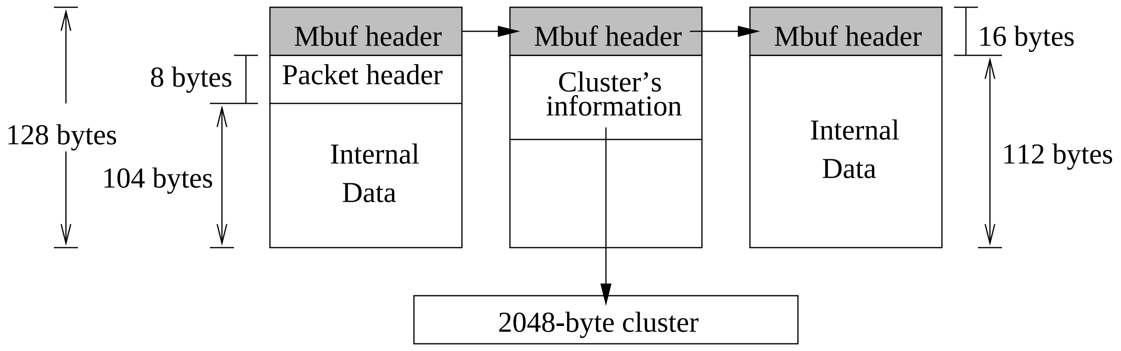


Figure 2.4: Example of an mbuf chain under IRIX 6.2

buffer size" in this thesis. We call the total amount of data being sent the "data size" (the data may be sent in multiple passes). On the IRIX 6.2 system we are using, the size of the data buffer is limited by the maximum value of a 4-byte signed integer. The data buffer size is specified by the application, and since it can affect how the application's data is fragmented, it can presumably affect the network throughput of the application as well.

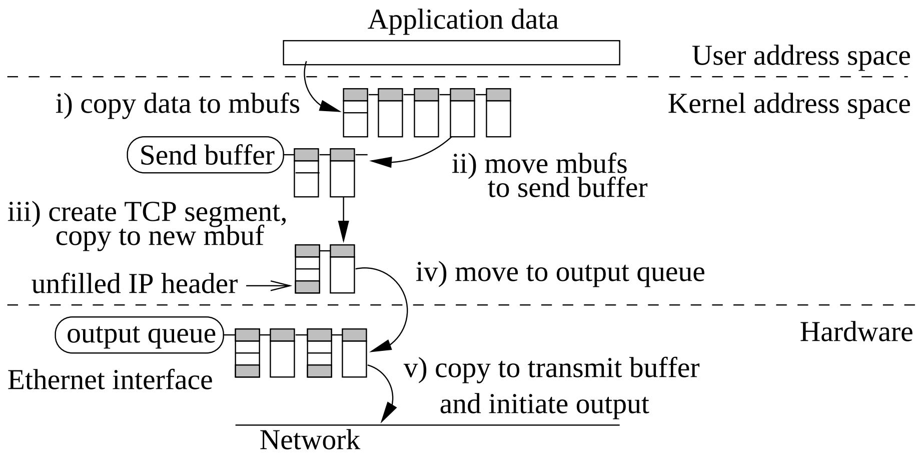


Figure 2.5: Network output under BSD Sockets

After the network output function is called, the kernel determines whether enough space is available in the socket send buffer. If so, the data is copied from the user address space into the required mbufs in the kernel address space (Figure 2.5-i) and appended to the send buffer (Figure 2.5-ii). For TCP, this is done one mbuf (or mbuf cluster) at a time. In our example, to store the 10 KB of data, a chain of 5 (10240/2048) mbufs, each referencing an external cluster, is required. If the send buffer becomes full, the send routine blocks until some data in the send buffer is sent to the network and enough space is available for one mbuf (or mbuf cluster, if required). For TCP, the mbuf or mbuf chain in the send buffer is deleted only after acknowledgment is received from the remote host. The use of a large send buffer size would reduce the chance of this blocking and presumably improve the network throughput, but it would also consume more kernel resources.

The TCP flow control algorithm [50] is then used to determine whether the network is ready to accept a TCP segment and if so, the size of that segment. The size of data in the segment cannot exceed the Maximum Segment Size (MSS) which is negotiated when the connection is established.

If the size of the data (still in the send buffer) is larger than the MSS, the data is sent in multiple segments. For TCP, the MSS is chosen to exactly fit into the data link layer's MTU, and on the Ethernet network this is 1460 (1500-20-20) bytes (see Figure 2.2). The data of size equal to one TCP segment is then copied from the send buffer into a new mbuf (or mbuf chain), by reference if stored in mbuf clusters. Another new mbuf containing a 20-byte TCP header and a 20-byte unfilled IP header (stored at the end inside the mbuf) is prepended to the mbuf (or mbuf chain), as shown in Figure 2.5-iii. For TCP, the original mbuf (or mbuf chain) in the send buffer is retained for possible retransmissions and is deleted after acknowledgment is received from the remote host.

One segment at a time is passed (by pointer) to the IP layer. The IP layer fills the skeletal IP header (at the end of the first mbuf in the chain, as shown in the figure), performs route selection, fragments the resulting IP datagram if necessary, and passes it down to the Ethernet device. Since the TCP MSS is set not to exceed the MTU, the TCP segments are sent without fragmentation.

The Ethernet device adds a 14-byte header before the IP header, and a pointer to the mbuf (or the first mbuf in the mbuf chain) is placed into the interface output queue for transmission (Figure 2.5-iv). A checksum is computed and appended to the packet by the Ethernet hardware during transmission. When the interface processes an mbuf that is on its output queue, it copies the data to its transmit buffer and initiates the output (Figure 2.5-v). Once the data is copied and a TCP acknowledgment is received from the receiver, the mbuf is released back into the kernel's pool of free mbufs.

Figure 2.6 shows how data is received using 4.4BSD Sockets. Upon receiving an input packet (Figure 2.6-i), the Ethernet interface generates a device interrupt. The Ethernet checksum is computed and checked by the interface hardware and is not passed to the kernel (the frame is dropped if the checksum fails). If a complete Ethernet frame has arrived, the frame is transferred to an mbuf chain. In our example, the 10 KB of data is sent in 8 such frames. The Ethernet device determines the type of protocol used by the received data by examining the Ethernet header, which is then removed. It demultiplexes (the IP packets are recognized by the "type" field inside the Ethernet frame), queues the received packet for processing, and schedules a software interrupt to invoke the appropriate input handling routine. For an IP packet, an IP input routine is invoked and the packet is placed on an IP input queue (Figure 2.6-ii). It is discarded if the queue is full. The default limit for the IP queue is 50 packets, which is set at system initialization time [59].

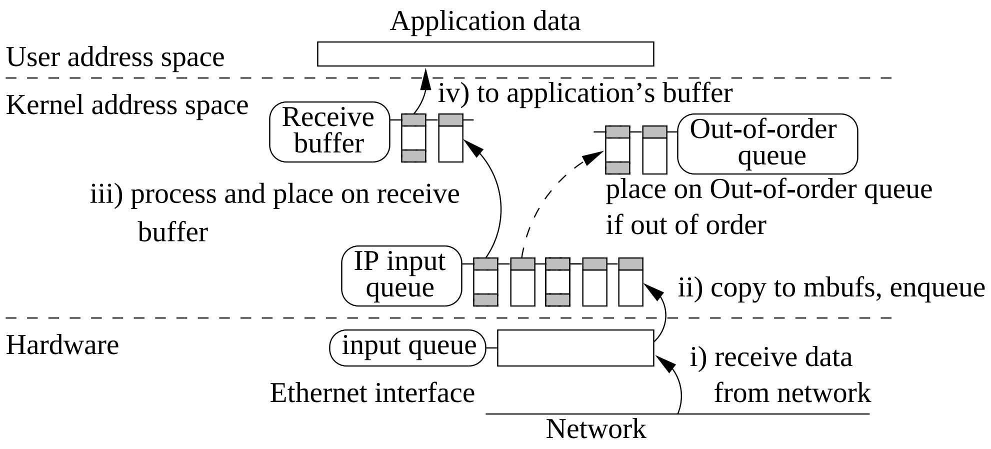

Application data
User address space
Kernel address space
Receive buffer
iv) to application's buffer
Out-of-order queue
iii) process and place on receive buffer
place on Out-of-order queue if out of order
IP input queue
ii) copy to mbufs, enqueue
Hardware
input queue
i) receive data from network
Ethernet interface
Network

Figure 2.6: Network input under BSD Sockets

During the software interrupt, the IP packets are taken from the IP input queue until the queue is empty. Verification is performed on each IP packet (e.g., IP checksum and packet length). Reassembly is not needed for TCP, since each IP packet contains a complete TCP segment. In the case of the 10 KB of data in our example, the IP layer passes 8 TCP segments to the TCP layer.

The TCP layer extracts the IP and TCP headers, and verifies the TCP segment. The packet header is then retrieved to identify the segment for reassembly. The segment is either appended to the receive socket buffer (Figure 2.6-iii) if it is in order, or placed into an "out-of-order queue" for future re-ordering. If the TCP_NODELAY flag is set, an acknowledgment is returned to the sender as soon as a TCP segment is received. If the flag is not set (this is the default), the acknowledgment is sent in a fixed time interval (this is 200 ms in most network implementations [59]).

The recipient application reads the data by invoking system calls such as `read()`, `readv()`, `recvfrom()` or `recvmsg()`. Internally, these system calls all invoke the same routine [57]. It blocks until the requested amount of data is present in the receive socket, or until some conditions are met (e.g., the timeout occurs, the non-blocking flag is set, or the receive socket buffer is full). The data is then copied from the kernel address space into the application's data buffer in the user's address space (Figure 2.6-iv). In the example, the 10 KB data is transferred from the receive socket buffer to the application's data buffer one mbuf at a time. The mbufs are freed after the data is copied. When the amount of data requested by the application has been copied, the network input routine returns control to the application. Since the data buffer size, which is defined in the receiver application, can affect the efficiency of retrieving the data from the receive socket buffer, it is also expected to impact the throughput of the application.

2.2.2 System V Implementation

Solaris 2.5.1 [29] is a System V-derived UNIX system. Unlike BSD systems, System V handles networking through the use of STREAMS. Although System V Release 4 (SVR4) [28] provides BSD Sockets compatibility at the API level, its internal networking architecture is STREAMS-based. It is therefore essential for us to understand the basic concepts behind STREAMS, which are described in this section.

The STREAMS facility provides full duplex paths between user processes and devices. Figure 2.7 shows a STREAMS implementation for TCP/IP over Ethernet. As can be seen in the figure, an individual stream has three components:

- a "stream head" provides the interface between the stream and user process,
- the optional "modules" (TCP and IP in the example in Figure 2.7) process data that travels between the stream head and driver, and
- a "stream driver" (Ethernet driver in the example) which may be a device driver or an internal software driver, converts data from the stream into a form suitable for the hardware device.

Each STREAMS module contains a pair of "message queues" — a read queue and a write queue. STREAMS perform network I/O by passing "messages" from one module's queue to that of another. Each of these messages consists of a list of data structures known as "message blocks"; each block carries data, as well as an embedded type field that identifies its type. The two most frequently used types are `M_DATA` and `M_PROTO`. The `M_DATA` message blocks contain application data, whereas the `M_PROTO` message blocks contain control information such as protocol headers.

When data is copied from a user's address space to the kernel address space, the stream head obtains a message block from the kernel. Under Solaris 2.5.1, a set of free message blocks with sizes defined at system initialization time are available for use. The kernel selects the message block

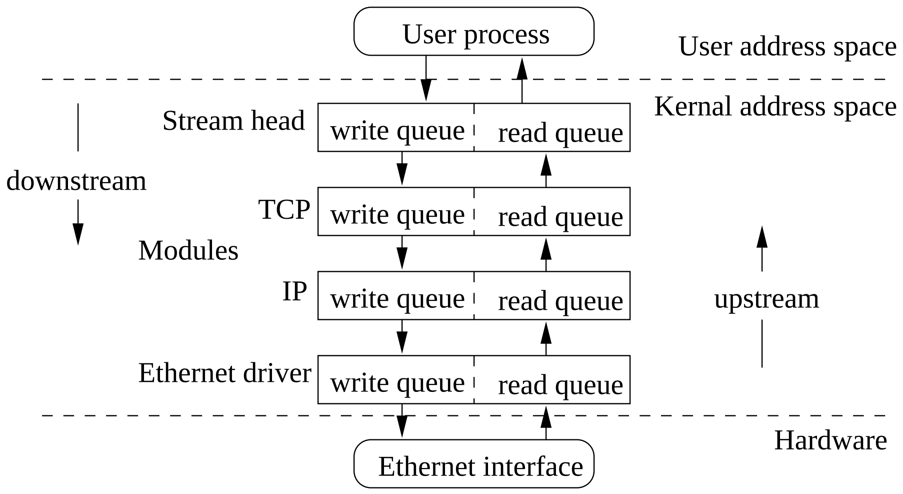


Figure 2.7: STREAMS implementation of TCP/IP over Ethernet

with the smallest size possible which has enough room to contain the message block. If the message block is too large, it is segmented into smaller blocks. Table 2.1 shows the pre-defined message block sizes selected for different data buffer sizes on the Sun platform of our testbed (these were obtained using of the `netstat` UNIX command with the `-m` and `-v` flags). The first column lists the ranges of data sizes passed down to the stream head, and the second column lists the message block size selected for use. For example, the table shows that 80 bytes of data can be stored in a 152 byte message block, whereas 81 bytes of data has to be stored in a 312 byte message block. This indicates that additional message block information (e.g., the message block header) takes up 72 bytes.

Data buffer size (bytes)	Message block size (bytes)
1 — 16	88
17 — 80	152
81 — 240	312
241 — 528	600
529 — 1008	1080
1009 — 1488	1560
1489 — 1904	1976
1905 — 2576	2648
2577 — 3952	4024
3953 — 9393	9464

Table 2.1: Message block allocations

Each STREAMS module provides a `put()` function and a `service()` function for its write and read queues respectively. It also has a flow-control mechanism to determine if its queues are ready for input. It contains separate upstream and downstream high-water marks and low-water marks that are used for flow-control calculation, and provides a `canput()` function for other modules to test its flow-control. A high-water mark limits the amount of data that may be outstanding in the queue, whereas a low-water mark specifies the minimum amount of data allowed in a queue.

Under Solaris 2.5.1, the default high and low water marks for both the sending and receiving of the TCP module are 8 KB and 0 bytes respectively (these values are obtained using the `ndd` UNIX command). Since these are system parameters that affect the whole system, it is not practical for us to alter them in our experiments.

A module passes messages to the next module by invoking the `put()` function of that module. Before doing so, the current module checks whether the queue of the next module is blocked (e.g., when the high-water mark is reached) by calling the `canput()` function of that module. If so, the message cannot be sent immediately, and is left on the current message queue, from where it may be processed again by its own `service()` function. This is periodically invoked by a STREAMS scheduler.

Figure 2.8 shows how sockets are implemented in SVR4 [65]. SVR4 does not truly implement BSD Sockets but provides sockets compatibility with a user-level library, *socklib*, and a STREAMS module, *sockmod*. The *socklib* library is used to map socket functions (`socket()`, `bind()`, `listen()`, `accept()`, `send()`, `recv()`, etc.) to STREAMS system calls and messages. The *sockmod* module mediates *socklib*'s interactions with the transport provider (TCP module in our case). Hence, the socket functions are no longer direct system calls under the SVR4 implementation. However, `read()` and `write()` are still system calls which do not require the *socklib* library, and they pass data directly to the stream head inside the kernel. In the rest of this thesis, we use "sockets" to refer to both BSD sockets and SVR4's emulation of sockets.

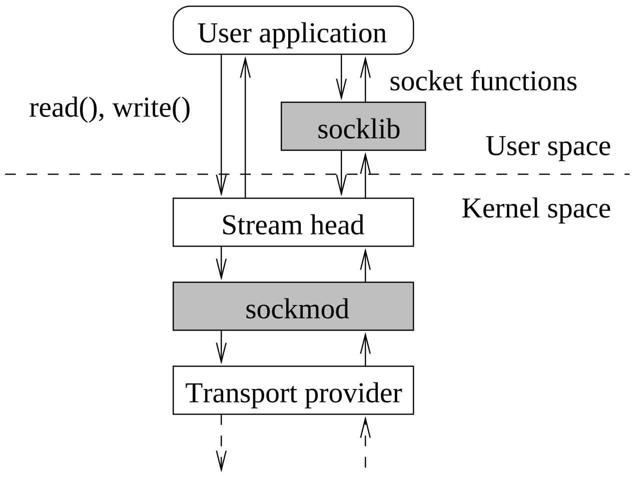


Figure 2.8: Sockets implementation under SVR4

For example, to send 10 KB of data using STREAMS over a TCP/IP/Ethernet network, an application invokes either the `write()` system call, or the `send()` or `sendto()` socket functions. In the case of the socket functions, *socklib* first looks up the socket's state and deals with any special actions that are required (such as the sending of out-of-band data). The `write()` system call is then invoked to copy the data into the kernel and place it on the stream head's input queue (if the data contains out of band data, a similar call to `write()` is invoked instead [65]). If the application invokes the `write()` system call, the data is directly copied into the kernel and passed to the stream head.

To copy the data into the kernel space, the kernel first allocates enough `M_DATA` message blocks for the data, which is fragmented into units of size equal to the TCP MSS. This is 1460 bytes for TCP over Ethernet (the Ethernet MTU is 1500 bytes, the IP and TCP headers are 20 bytes each), so each unit fits into a 1560-byte message block (from Table 2.1, this is the smallest message block that can accommodate the entire TCP MSS). The 10 KB data is copied, one message block at a time [28], into 8 ($10240/1460 \leq 8$) such blocks. The flow-control mechanism of the stream head applies (e.g., the transfer is blocked if the stream head's input queue is full). The stream head then sends them down stream to the *sockmod* module, which checks the socket information of each message. A TCP message is simply passed downstream to the TCP module.

The TCP module passes the message downstream to the IP module, then to the Ethernet driver, and eventually to the hardware. The `M_PROTO` message blocks containing the TCP and IP protocol headers, as well as the Ethernet header and trailer are added to the message during the process. Message blocks in the kernel are copied by reference (i.e., only pointers to the message blocks are copied).

At the receiving end, the `read()` system call or the socket functions `recv()` or `recvfrom()` is invoked by the recipient application. It is blocked until the data is available in the kernel. When the data arrives at the STREAMS driver from the network interface, it is stored in message blocks which travel upstream, with `M_PROTO` message blocks containing the appropriate protocol information being removed at each module, until the data reaches the stream head (the 10 KB data is stored in 8 message blocks). Again, message blocks in the kernel are copied by reference. These message blocks are then copied into the specified buffer of the recipient application in the user address space. The data is copied from the kernel to the application's data buffer one message block at a time. The kernel then returns control to the application.

2.2.3 Time Traces of Network Input and Output

In order to illustrate BSD sockets and STREAMS network implementations previously discussed in Sections 2.2.1 and 2.2.2, we have written a simple program, which transfers a specified amount of data between a client and a server and records a time stamp every time an input or output function is called on either side.

In order to prevent TCP segmentation and hence simplify our analysis, we choose a 1460 byte data buffer (equal to the TCP MSS). We send a total of 146000 bytes of data in each transfer, so that each side of the transfer must invoke its network I/O call exactly 100 times in order to send or receive all of the data (i.e., 100 time stamps are recorded on each side). The default socket buffer sizes are used. A graphic representation of the times recorded is shown for the SGI's (BSD-based) in Figure 2.9-(a) in the client-to-server direction, and Figure 2.9-(b) in the server-to-client direction. The same is shown for the Sun's (STREAMS-based) in Figure 2.10-(a) in the client-to-server direction, and Figure 2.10-(b) in the server-to-client direction.

In each graph, the x-axis represents the 100 time stamps recorded from each side in the same data transfer, and the y-axis shows the elapsed time of each time stamp. The elapsed time is calculated based on an initial time stamp "0", which is recorded immediately after the connection is established. Thus, the elapsed time of time stamp "0" is always 0 seconds in the graphs. For example, time stamp "100" in Figure 2.9-(a) is recorded 0.34 second after time stamp "0".

On both platforms, the sender is blocked for a short time (breaking the lines into segments in the graphs) after a certain number of network input or output calls is issued. For example in the client-to-server direction on the SGI's, it is blocked after 42 network output calls are issued (as shown in Figure 2.9-(a)). This is when $42 \times 1460 = 61320$ bytes are sent, indicating that the sender is blocked when the send socket buffer is full (recall that the SGI's running IRIX 6.2 use a default socket buffer size of 61440 bytes). As the data arrives on the other side, and TCP acknowledgments are received by the sender, space is freed in the send socket buffer, and this is when the blocking

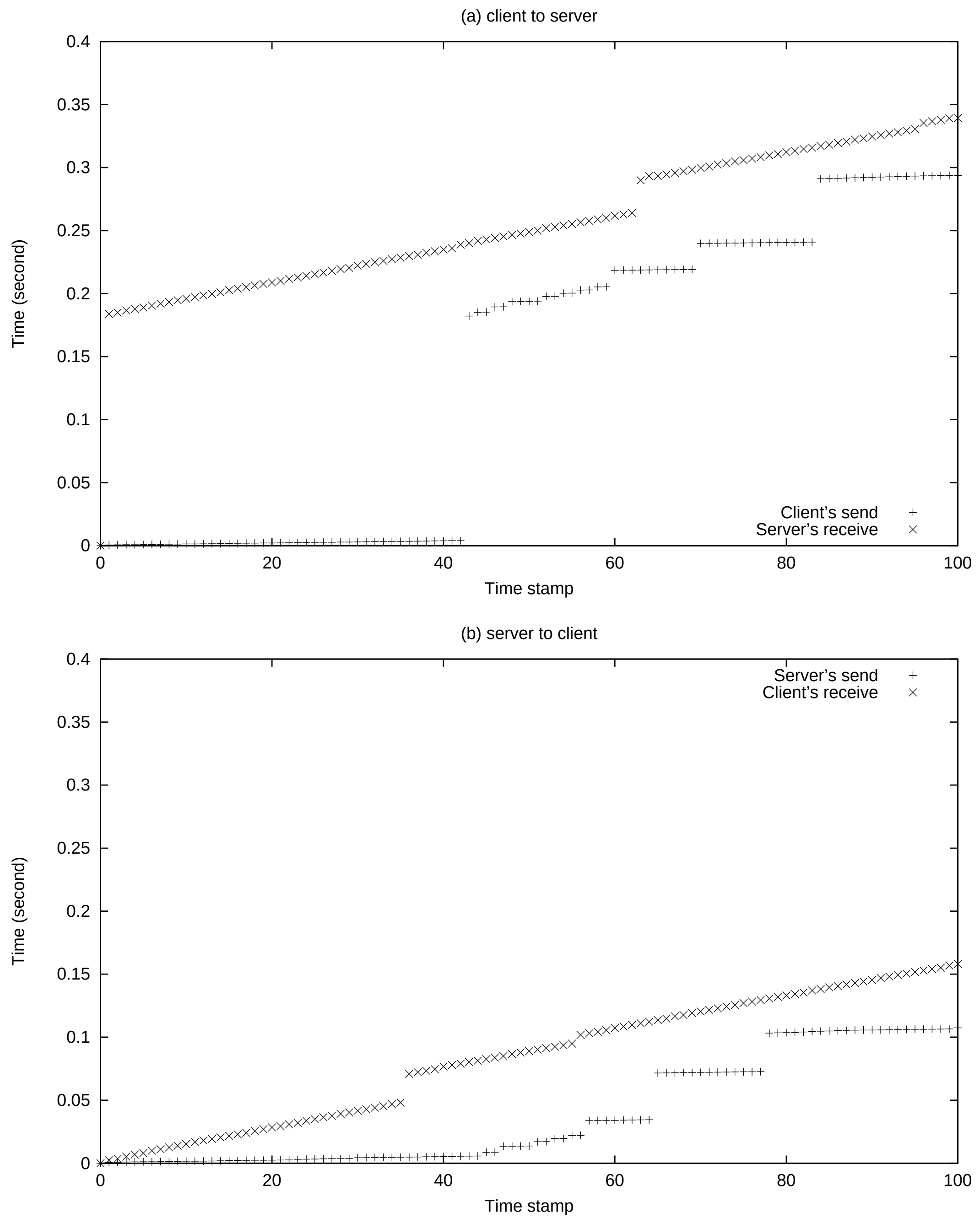
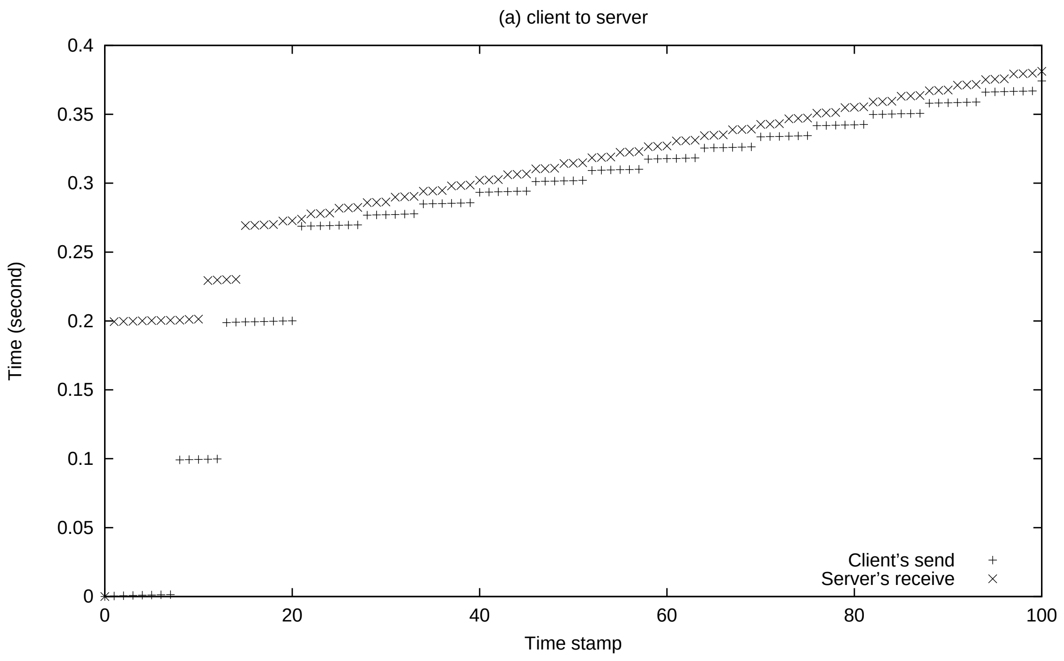


Figure 2.9: Time traces of Network input and output (SGI's)



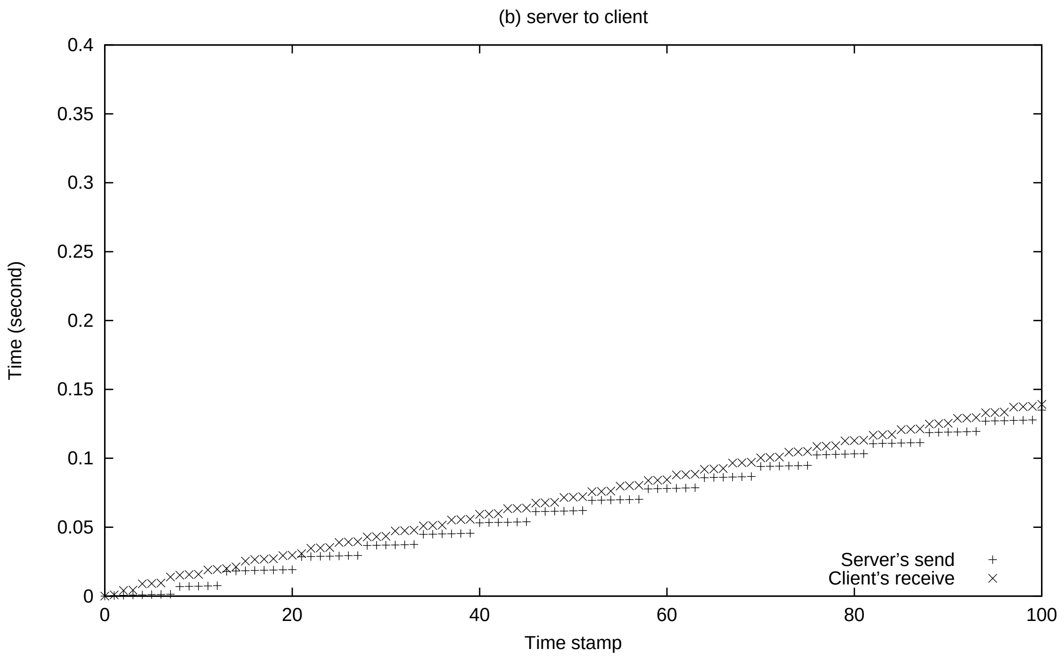


Figure 2.10: Time traces of Network input and output (Sun's)

ends and additional data can be sent (from time stamp "43" onwards in the figure). Note that in the opposite direction (Figure 2.9-(b)), the blockage is not obviously seen, and occurs a few calls after the 42nd call. This is because the client receives data as soon as the connection is established (for reasons unclear to us but discussed in more detail later, this does not happen in the client-to-server transfer direction), and some space in the sender's send socket buffer is freed before the 42nd network output call is issued.

A similar pattern is observed on the Sun platform, but the blockages occur more often — in general, once every 6 calls for sending (when $6\times1460=8760$ bytes are sent), and once every 3 calls for receiving (when $3\times1460=4380$ bytes are received). While the former confirms the use of a default 8 KB high-water mark by the TCP module in our systems, the latter suggests that some kind of bottleneck is encountered in receiving data (possibly a smaller high-water mark used in another module).

We observe that the sending time is faster than the receiving time in all cases (the transfer time on each side is equal to the elapsed time of the last time stamp, i.e., time stamp "100", of that side). This is because the sender's network output function returns before the data is received by the receiver. This suggests that the transfer time of the sender may not accurately reveal the actual data transfer time. Therefore, we should either measure the transfer time from the receiver's side, or apply some kind of technique to ensure that the sender waits for the receiver to receive all of the data before stopping (as is done in Netperf, see Section 3.2).

We also observe on both platforms that, in the client-to-server transfer direction, the server waits for 200 ms after the connection is established before it starts to receive any data; this does not happen in the opposite transfer direction. Using the `TCP_NODELAY` option does not eliminate the anomaly. The exact reason for this delay is unknown to us.

2.3 Fore Systems ATM Environment

This section introduces the ATM equipment used in our testbed and provides a brief overview of the Fore ATM network implementation.

2.3.1 The ATM Hardware

Our experiments are conducted using a non-blocking *ForeRunner* ASX-200BX ATM Switch with an aggregate switch bandwidth of 2.5 Gbps [24]. The Sun Sparcstation5's and SGI Indy's in our testbed are equipped with the SBA-200E [22] and GIA-200E [25] adapter cards, respectively . Both cards utilize an embedded Intel i960 RISC processor, along with special-purpose AAL5 and AAL3/4 segmentation and reassembly (SAR) sublayer (see Section 2.1.4), and Direct Memory Access (DMA).

2.3.2 The ATM Software

The Fore Systems' software Forethought 4.1.0 is used in our testbed. It provides two alternative paths to the ATM hardware: either traditional TCP/IP based sockets can be used over ATM, or the Fore System's API can be used to interface with the ATM hardware directly using a data link layer (OSI layer 2) interface. This environment is shown in Figure 2.11 [24].

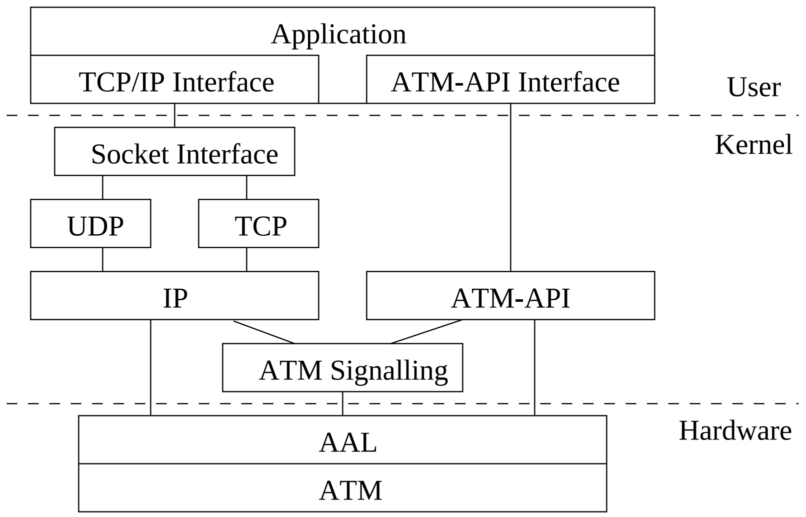


Figure 2.11: Fore System's software/hardware environment [24]

2.3.3 Fore System's Application Programming Interface

The Fore System's Application Programming Interface (Fore API) offers a set of user-level library routines for writing native ATM applications. This socket style abstraction interacts with the native ATM application at the data link layer. Hence, the transport layer semantics of sockets are used with data link layer primitives. Section 3.1 explains these routines in more detail.

2.3.4 The Fore ATM Network Implementation

The Fore API for both Solaris and IRIX platforms are STREAMS-based [41], and communicate with the ATM device driver at the data link layer using the Data Link Provider Interface [63], which is an interface standard for STREAMS modules to communicate at the data link layer. Since the Fore API is propriety, we do not know the details of its implementation.

The version of the Fore Systems' software which we use assigns each message block a size of 4 KB in IRIX 6.2 [25]. The size assigned in Solaris 2.5.1 is unfortunately not known to us. We give an example of how a 10 KB message is transferred on the SGI's using AAL5 from a native ATM application on one host to another on a remote host.

The application begins by calling the `atm_send()` function call. The data is then passed, one message block at a time, to the STREAMS head inside the kernel. A 4 byte CPCS-PDU header is prepended to the CPCS-PDU. The ATM driver then instructs the ATM adapter card to perform the necessary SAR (see Section 2.1.4) and sends the data (already packaged in ATM cells). The ATM switch performs the basic routing routines to send the data to the destination host.

At the receiving end, the recipient application calls the `atm_recv()` function to receive the data, and is blocked until the data is available in the kernel. As data arrives from the network, the ATM adapter card performs SAR on the ATM cells to form the CPCS-PDU. The adapter card uses a pool of buffers belonging to the kernel — called "loaned buffers" which have a maximum size of 4 KB in IRIX 6.2. The size assigned in Solaris 2.5.1 is not known to us. Using Direct Memory Access, it puts the incoming CPCS-PDU into these buffers and interrupts the kernel. The kernel passes these buffers upstream by reference (i.e., only passing the memory addresses, so there is no need for copying). The `atm_recv()` is then informed by a kernel interrupt that the data is available, and the data is copied to the user space one message block at a time. The buffers are then returned for use by the ATM adapter card.

2.4 TCP/ATM versus Native ATM networks

ATM networks are inherently incompatible with traditional networks. A number of strategies to support legacy network protocols over ATM networks are continuously being proposed and discussed by the networking community. Many of these, such as LAN Emulation (LANE) [6], Classical IP over ATM [38] and Multiprotocol Over ATM (MPOA) [7], involve providing mechanisms to interoperate IP with the ATM layers and hiding the ATM network from the layers above IP. These are commonly known as "IP over ATM" (or IP/ATM). When TCP is used on top, it is called "TCP over ATM" (or TCP/ATM).

Since legacy network protocols were not originally designed for ATM networks, emulating them over an ATM network requires deliberately hiding some of the features of ATM networks. The additional protocol layers require extra headers, trailers and redundant checksum calculations, and this has been shown to increase latency [67]. Moreover, IP address resolution, while not required in ATM networks, is retained solely for compatibility.

Despite the on-going development of TCP/ATM, some developers have considered eliminating the intermediate protocols and writing raw ATM applications using the ATM API. Such applications are called "native ATM applications". However, due to the amount of work required to port applications to native ATM interfaces, and due to the lack of standard API, native ATM applications are not common.

This section describes the Fore TCP/ATM implementation and the characteristics of TCP/ATM. Related work on the performance of TCP/ATM and native ATM is also presented.

2.4.1 Classical IP and Fore IP

The Fore ATM environment we use is capable of transmitting IP packets through the ATM network and mapping IP addresses onto ATM addresses for IP/ATM. Two implementations are supported: Classical IP (or Classical IP over ATM) [38] is a standard developed by IETF, and Fore IP [23] which is a propriety Fore Systems approach. We only use Fore IP in our experiments because we believe that it introduces less protocol overhead. In order to show this, we provide a comparison between the two implementations.

Classical IP encapsulates IP packets using the IEEE 802.2 LLC/SNAP encapsulation scheme [30], in which an 8-byte header is prepended to each IP packet to form the CPCS-PDU, which is then segmented into ATM cells under AAL5 (see Section 2.1.4). In this thesis, we call the maximum size imposed on the CPCS-PDU the "ATM MTU". Figure 2.12 shows this encapsulation scheme using the default 9180-byte ATM MTU [5]. The size of the padding is 20 bytes, allowing the resulting CPCS-PDU to be divisible by 48 bytes.

Fore IP allows communication using AAL5 (or AAL3/4) with no encapsulation (unlike Classical IP, Fore IP uses a broadcast address resolution protocol [23], saving the 8-byte header). Thus, the default ATM MTU size for Fore IP is 9188 bytes [24]. Figure 2.13 shows the Fore IP encapsulation using a 9188-byte ATM MTU size. The maximum ATM MTU allowed in Fore IP is 65535 bytes. Excluding the 4 byte CPCS-PDU header (see Section 2.3.4), an application can send up to 65531 bytes of data at one time.

Since Fore IP does not require the 8-byte LLC/SNAP header, and therefore introduces slightly less protocol overhead, we use it for our studies.

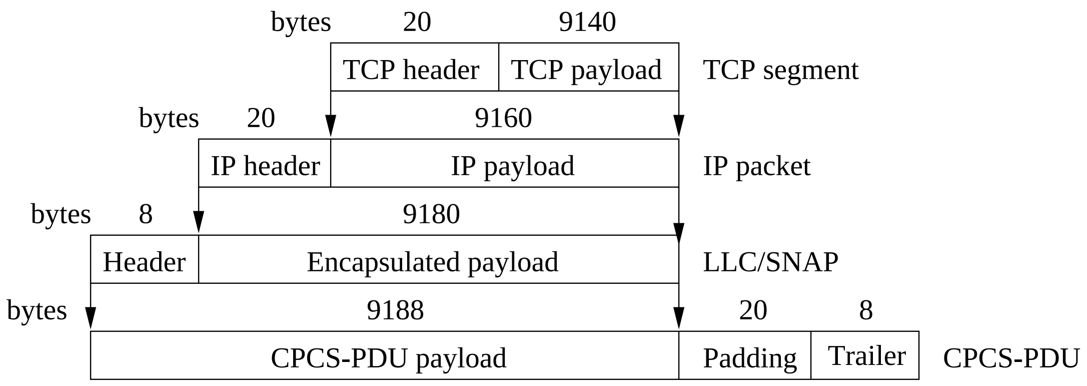


Figure 2.12: LLC/SNAP encapsulation of IP packets

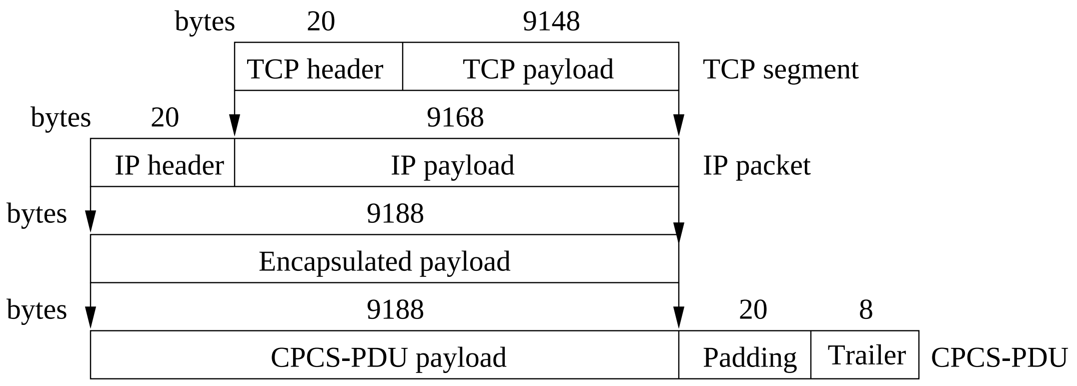


Figure 2.13: Fore IP encapsulation of IP packets

2.4.2 Characteristics of TCP/ATM Networks

MTU and MSS

A mathematical study of data loss with respect to the probability of TCP fragment loss [35] concludes that, in a TCP/IP/Ethernet network environment, larger MTU sizes tend to reduce the chance of data loss, and hence give better performance. It is likely that the same is true for the TCP/ATM environment, which uses a rather large MTU size (9188 bytes by default, with a maximum of 65535 bytes) compared to the Ethernet's 1500-byte default MTU.

Under the TCP protocol, the MSS size is adjusted to fit into the MTU [50, 57]. For the TCP/ATM environment using Fore IP, our Sun's use a default and maximum MSS of 9148 (9188-20-20) bytes and 65495 (65535-20-20) bytes, respectively. The BSD network implementation applies an additional rounding algorithm to reduce the adjusted MSS to the next lower multiple of mbuf clusters [59]. Thus, the default and maximum MSS's on the SGI's are reduced to 8 KB and 60 KB, respectively (in IRIX 6.2, an mbuf cluster is 2 KB).

Protocol Overhead

Each protocol layer imposes overhead as it adds additional headers or trailers to the data packets. The header and trailer do not contain any transmitted data, yet they consume network bandwidth, and as a result are often referred to as "protocol overhead".

By evaluating, on a layer-by-layer basis, the portion of a data packet actually used in this manner, one can obtain a theoretical upper-bound on the bandwidth available in any TCP/ATM network. For example, we consider the protocol overhead imposed by the TCP/IP/ATM/AAL5 protocol stack using Fore IP with an 9188 byte ATM MTU (ignoring the physical layer). In Figure 2.13, we see that a 9148-byte TCP segment results in a 9216-byte CPCS-PDU. Since an ATM cell comprises a 5-byte header and 48-byte payload, this CPCS-PDU requires 192 (9216/48) ATM cells, resulting in an additional 960 (192×5) bytes of ATM header information. Therefore, only about 89.9% (9148/(9216+960)) of the network bandwidth is actually used for transmitting data. If different physical layers are considered, this can be reduced to 87% or less of the total bandwidth of the ATM network [13]. In fact, most of the overhead is shown to be consumed by the ATM and AAL layers [13, 1, 41].

Thompson *et al.* [61] perform an extensive study of the Internet traffic captured with the internetMCI and vBNS backbones. They observe a "bit-efficiency" of approximately 80% (i.e., 80% of the total bandwidth is utilized for user data) for IP/ATM traffic using AAL5 and LLC/SNAP (Figure 2.12) encapsulation in real systems.

The factor of protocol overhead potentially gives native ATM an advantage since theoretically less overhead is introduced compared with TCP/ATM. However, the presence of other overheads, such as the operating system and memory copying costs described by Clark *et al.* [16], are also expected to have significant impact on the overall throughput, thus preventing the applications from obtaining throughputs equal to the available bandwidth.

Throughput Deadlock

A result from our TCP/ATM experiments (see Section 4.3.2) shows an unexpectedly low throughput when using a certain combination of socket and data buffer sizes. We believe this is caused by the "throughput deadlock" problem.

In a number of studies designed to measure throughput over TCP/ATM networks [17, 45, 40], an anomalous behavior is encountered in which throughput drops dramatically for certain combinations of socket buffer sizes. It is agreed [17, 45, 40] that this is caused by Nagle's Algorithm (also known as Silly Window Syndrome Avoidance) [15] together with the receiver side delayed acknowledgment algorithm [15]. Deadlock occurs when the sender waits for an acknowledgment from the receiver, while the receiver waits for more data from the sender. The transfer resumes only when the timer issues a time-out after 200 ms. This occurs in TCP/ATM because of the large MTU size used. Using simple calculations, Comer and Lin [17] conclude that using a send socket buffer size no smaller than 3 times the MSS (with any receive buffer size) would prevent such anomalous behavior.

2.4.3 Performance of TCP/ATM and Native ATM Networks

Since there are a great number of network configurable software and protocol parameters whose effect on throughput can be studied, it would be intractable to test all combinations of these parameters in our experiments. Fortunately, there are several studies [40, 26, 41, 1] that have performed throughput measurements using the TCP/ATM and native ATM network environments, and which have testbed configurations and methodologies similar to ours. Although none of these studies has used real network applications, we believe that they do provide insights into those parameters which can have potential impact on throughput. The results of these studies are presented in this section.

Tables 2.2 and 2.3 summarize the maximum throughputs obtained from these studies using the TCP/ATM and native ATM environments, respectively. Each column in the tables shows the configurations and parameters used for each study. The first row lists the type of end-stations, the second row lists the operating systems, and the third, fourth and fifth rows list the network configurations used. The second-to-last row shows the data buffer sizes used to achieve the maximum throughputs and the last row shows the maximum throughputs obtained. The benchmark used is shown in the sixth row of Table 2.2 and seventh row of Table 2.3. The seventh row of Table 2.2 shows the socket buffer size used to obtained the maximum throughputs. The sixth row of Table 2.3 shows the AAL type used.

Study (TCP/ATM)	Luckenbach *et al.* [40]	Fouquet *et al.* [26]	Matijašević *et al.* [41]	Andrikopoulos *et al.* [1]
End-stations	Sparcstation10	Sparcstation10	Sparcstation10	Sparcstation20
Operating system	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)	Solaris 2.4
ATM switch	ASX-100	ASX-100	ASX-100	ASX-200
Adapter card	SBA-200	SBA-200	SBA-200	SBA-200
Physical interface	TAXI (100 Mbps)	TAXI (100 Mbps)	TAXI (100 Mbps)	OC-3 (155 Mbps)
Benchmark	ttcp	Netperf	Netperf	Netperf
Socket buffer size	52428 bytes	8192 bytes	52428 bytes	18296 bytes
Data buffer size	N/A	4096 bytes	65536 bytes	64036 bytes
Throughput	**59 Mbps**	**21 Mbps**	**49 Mbps**	**94 Mbps**

Table 2.2: Maximum TCP/ATM throughputs from various studies

Luckenbach *et al.* [40] conduct tests on their network, which consists of a Fore ASX-100 ATM switch and several Sparcstation10's connected to the switch using a 100 Mbps TAXI interface. The major objective of their work is to compare the SBA-100 and SBA-200 SBus adapter cards. They highlight the "saw tooth" phenomenon — a periodic rise and fall in throughput — observed as the CPCS-PDU size increases, with a distance of 48 bytes between the peaks. This is due to the influence of CPCS-PDU padding (Figure 2.3). The peaks of the saw-tooth structure are observed

Study (native ATM)	Luckenbach *et al.* [40]	Fouquet *et al.* [26]	Matijaševič *et al.* [41]
End-stations	Sparcstation10	Sparcstation10	Sparcstation10
Operating system	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)	SunOS 4.1.3 (Solaris 1.0)
ATM switch	ASX-100	ASX-100	ASX-100
Adapter card	SBA-200	SBA-200	SBA-200
Physical interface	TAXI (100 Mbps)	TAXI (100 Mbps)	TAXI (100 Mbps)
AAL	AAL5	AAL5	AAL5
Benchmark	N/A	Netperf	Netperf
Data buffer size	4096	4096 bytes	4092 bytes
Throughput	**83 Mbps**	**62 Mbps**	**84 Mbps**

Table 2.3: Maximum native ATM throughputs from various studies

when no padding occurs, that is, when the CPCS-PDU payload size is 8 bytes less than a multiple of 48 bytes (8 bytes are used for the trailer, see Section 2.1.4). With the SBA-200 cards, AAL5 on a uni-directional (simplex) connection, and an ATM MTU size of 4096 bytes (the maximum size allowed by their ATM driver), a maximum throughput of 83 Mbps is achieved. There is no clear indication of the measurement tool used. Other Fore IP TCP/ATM experiments use the public domain benchmark program TTCP [64] and show that throughputs increase with the socket buffer sizes, and that the maximum throughput of 59 Mbps is obtained when the sizes of both send and receive buffers are set to their maximum possible values (52428 bytes in their operating system).

Note that the "saw tooth" phenomenon can only be observed if the data buffer size is incremented in small intervals (less than 48 bytes). Since we use larger intervals (at least 1 KB) throughout our experiments, we do not observe this phenomenon in our results.

Fouquet *et al.* [26] conduct native ATM and TCP/ATM tests on two Sparcstation10's with SBA-200 SBus adapter cards and a Fore ASX-100 ATM switch, connected using a 100 Mbps TAXI interface. They use the public domain network benchmark, Netperf [21], as well as two other programs they wrote, one for memory transfer and another for file transfer. All three programs utilize the Fore API and AAL5 for native ATM. They conduct "loopback tests" — using a single machine as both the source and destination — and define their results as "architectural limits", which are free from network bottlenecks. For native ATM experiments, they use data sizes of 1, 5, 10 and 15 MB and data buffer sizes of up to 4096 bytes (the maximum their ATM driver allows). TCP/ATM experiments are done using socket buffer sizes of 8 KB and data buffer sizes of up to 4096 bytes (there is no clear indication of the data sizes used). By comparing the results obtained from machines connected back-to-back without the ATM switch, they conclude that the switch does not have any observable effect on throughput. Cell loss occurs in memory transfers but not file transfers. They believe that delays due to disk access times may have prevented the ATM switch buffer from overflowing. Their results also show that throughput increases with data buffer size, with no observable difference when using data sizes of 1 MB, 5 MB and 10 MB. However, the 15 MB data size introduces additional delays due to virtual memory swapping. Both native ATM and TCP/ATM results confirm the "saw tooth" phenomenon observed by Luckenbach *et al.* [40] when changing data buffer sizes. For TCP/ATM, memory transfers achieve a maximum throughput of 21 Mbps (28% of the architectural limit of 75 Mbps), and 15 Mbps (52% of the architectural limit of 29 Mbps) for file data transfer. When compared with the TCP/ATM results in the Luckenbach *et al.* study [40] (Table 2.2), we believe the rather low TCP/ATM throughput observed by Fouquet *et al.* [26] may result from the use of the comparatively small socket buffer (and possibly a small data buffer as well). For the native ATM environment, memory transfers achieve a maximum throughput of 62 Mbps (83% of the architectural limit of 75 Mbps) and file data transfer 25 Mbps (86% of the architectural limit of 29 Mbps). From these results, we can conclude that native ATM out-performs TCP/ATM, which looks promising for network applications.

Note that Netperf and both the programs written by the Fouquet *et al.* have no specific routine to guarantee reliable transfers, and there is no discussion of the reliability of native ATM transfer mentioned in the study. Indeed, cell loss is reported in their results. In addition, only four file sizes (1 MB, 5 MB, 10 MB, and 15 MB) are used in their study, and none of these is appropriate to represent a small file. Files as small as 500 bytes and 5 KB are used in our experiments, and they yield unexpectedly low throughputs.

Matijašević *et al.* [41] use Netperf to conduct TCP/ATM, UDP/ATM (UDP over IP/ATM) and native ATM tests on two Sparcstation10's with Fore SBA-200 SBus adapter cards and a Fore ASX-100 ATM switch using Netperf. A 100 Mbps TAXI interface is used as the physical layer. TCP/ATM experiments using Fore IP are conducted using various socket buffer sizes and data buffer sizes. A maximum throughput of 49 Mbps is observed when the socket buffer sizes are set to the maximum 52428 bytes (this result is less than the 59 Mbps obtained by Luckenbach *et al.* [40], which we believe is caused by the use of different data buffer sizes). Throughput increases with data buffer size, but it rapidly reaches a maximum (at a data buffer size of about 4 KB) and levels out. The native ATM experiments also show that throughput increases with data buffer size. A maximum throughput of 84.71 Mbps is observed using the maximum 4092 byte data buffer (Table 2.3).

Andrikopoulos *et al.* [1] use Netperf to examine the TCP/ATM throughput achieved on a variety of platforms using Fore SBA-200 cards and a Fore ASX-200 ATM switch connected by 155 Mbps OC-3 links. They conduct "loopback tests" to obtain what they call "architectural limits", and observe the maximum throughput of 93.91 Mbps (Table 2.2) using Fore IP between two Sparcstation20's. This is about 70% of its theoretical maximum of 134.6 Mbps. This peak occurs when the socket buffer size is twice the MSS (socket buffer size is varied at intervals of 1 MSS, so the peak may not be at exactly 2 MSS).

Almost all of these studies involve conducting raw data transfer experiments from the memory of an application on one host to that of another host, and none involves experiments with real network applications. Although the results show that native ATM out-performs TCP/ATM in all the raw data transfer experiments, we have no evidence that this is also the case when real network applications are used. It is therefore necessary to examine the performance of the real network applications.

After considering the results of these studies, we decided to choose a few network configurable software and protocol parameters to study their impact on throughput. They include the data buffer size, socket buffer size, and ATM MTU size. Different file sizes are also tested in our experiments.

2.5 Network Applications

The HTTP and FTP protocols have been selected for our experiments because they constitute what are considered to be the most frequently used Internet applications [27, 46, 61]. For example, in a 1996 study by Newman *et al.* [46], a traffic trace was obtained from an Internet backbone, containing five minutes of traffic obtained by monitoring an FDDI ring that connects traffic from the San Francisco Bay Area to and from an Internet backbone. It shows that, at that time, HTTP packets accounted for 40%, and FTP packets 12%, of all network traffic, making them the first and second most frequently seen types of packets. It is extremely likely that the use of these protocols has increased substantially since the time of this study, especially HTTP.

This section begins with an overview of the FTP and HTTP protocols, followed by an introduction to the network applications used in our experiments. Each is also described in more detail in Chapter 3.

2.5.1 Application Layer Protocols

HyperText Transfer Protocol

The Hypertext Transfer Protocol (HTTP) [9] is an application layer protocol for distributed, collaborative, hypermedia information systems. Originally developed at CERN in Switzerland, the protocol is widely used by the World Wide Web global information service.

HTTP is based on a request/response paradigm. All HTTP transactions take place over a TCP/IP connection (usually using the default port "80"). An HTTP transaction consists of four phases:

- *Connection* — the client attempts to connect with the HTTP server.
- *Request* — once the connection is established, the client sends a request to the server.
- *Response* — the server responds with the requested data, if any, and other necessary information.
- *Close* — after the server sends the response, it closes the connection.

HTTP messages consist of requests from a client to a server and responses from the server to the client. Requests and responses are ASCII strings as defined by the Telnet Protocol [51, 57].

The original protocol (HTTP/0.9) has only simple request and response formats. All our HTTP experiments use HTTP/1.0 [9], which is the current standard, and which allows more information to be included in the messages, such as the type and size of the message being returned. HTTP/1.1 [10] is proposing to extend the life-span of the TCP connections, and is in its last stage of development at the time of writing.

File Transfer Protocol

The File Transfer Protocol (FTP) [52, 57] was defined in 1985. It provides facilities for transferring files to and from remote computer systems. FTP operates in a client/server fashion and uses two TCP port connections — one as a command channel and one for data transfer — allowing it to simultaneously issue commands and transfer data between the client and server.

Under normal operation the FTP server listens on the well-known port "21" for control connection requests. The choice of port numbers for the data connection depends on the commands issued on the data connection.

At the start of an FTP session, the control connection between client and server is established, following the conventions of the Telnet Protocol [51, 57]. This connection remains open during the entire session and is closed by the server at the user's request after all transfers and replies are completed. The client sends FTP commands and interprets replies received. The server interprets commands, sends replies, and when necessary, sets up data connections and transfers data. FTP commands and replies are ASCII strings as defined by the Telnet Protocol [51, 57].

The client listens on the data port prior to sending a transfer request command. The client's default data port is the same as the control connection port. The server's default data port has a port number one less than that of the control connection port (i.e., port “20”). The FTP request command determines the direction of the data transfer. Upon receiving the transfer request, the server initiates the data connection to the port. When the connection is established, the data transfer begins and the server sends a confirming reply to the client. The connection is then closed by the server. Figure 2.14 depicts a typical FTP client/server scenario.

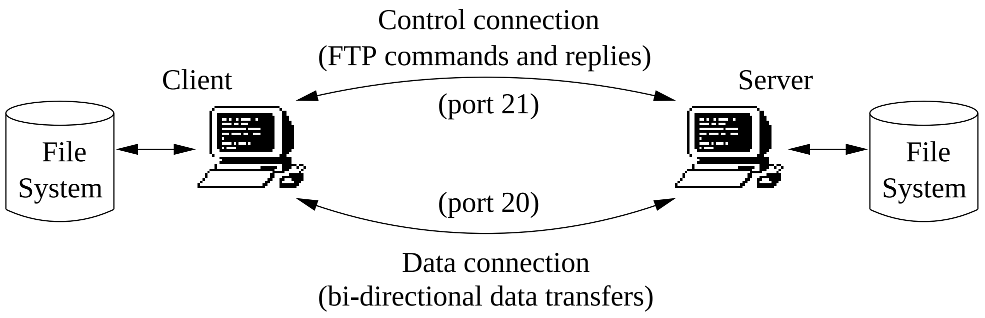


Figure 2.14: A typical FTP client/server scenario

2.5.2 Network Applications

Network Benchmarks

Netperf [21], developed at Hewlett-Packard, is a networking performance tool widely used in the computer industry as a standard network benchmarking tool. It is capable of performing bulk data transfers and request/response tests on a variety of platforms. It is also capable of supporting both TCP using sockets and native ATM using the Fore API (but no reliability scheme is implemented).

The Fore API test of the current version of Netperf does not support throughput measurement based on fixed sizes of data, which is required in our reliable native ATM protocol (see Section 3.1.3 for more details). In addition, Netperf uses a sophisticated transfer mechanism to maintain the accuracy of the measurements among different platforms, and this makes any modification difficult. Test TCP (TTCP) [64], created at the US Army Ballistic Research Lab (ARL), is designed for overall performance testing of bulk data transfers using TCP or UDP between two systems on IP-based networks. The TTCP program provides the ability to perform file-based data transfer tests; Netperf does not provide this ability. In addition, it allows a fixed size of data to be transfered in each test, and thus, is suitable for implementing our reliable native ATM protocol. However, a major disadvantage of TTCP is its incompatibility among different platforms, making it unsuitable as a standard benchmark.

We select Netperf for our memory-to-memory transfer experiments (Section 4.3) because it is a widely accepted standard benchmark supporting both BSD sockets and the Fore API. The TTCP benchmark, on the other hand, is chosen for our file access experiments (Section 4.4), as well as for the memory-to-memory transfer experiments using our reliable native protocol, for reasons we have discussed.

The Internet Daemon

The Internet daemon (inetd) is a UNIX utility program that listens to a range of ports on behalf of a large number of client applications. When a request to a particular TCP port is received, the application corresponding to that port is started by the inetd in order to service the request.

It is often believed that an inetd-based server will spend a significant amount of time starting and initializing (forking and exec-ing) a particular network daemon for each request. A study by McGrath [43] shows that using inetd on HTTP daemons greatly reduces the overall throughput. In his results, using inetd achieves no better than 38% of the throughput of the standalone version of the identical daemon. Indeed, most current UNIX-based Web server daemons have the ability to be run in standalone mode. However, most UNIX-based FTP daemons, including the Wu-ftpd we use in this thesis, still require the inetd. Unfortunately, we have not encountered any study which deals with the FTP performance using the inetd.

In order to make fair comparisons between the original applications and their native ATM counterparts, we decide to run our HTTP experiments in standalone mode and our FTP experiments via the inetd. In addition, we do not include the time to create an ftpd when measuring FTP throughput.

HTTP Server Benchmark and Client

The NCSA HTTP daemon [36], developed at the National Center for Supercomputing Applications at the University of Illinois, is used in our experiments. It supports the "pre-forking" design (see Section 3.4.1) which has been shown to improve performance [43] and enables it to out-perform other traditional HTTP servers [42]. Its code, however, is more complicated due to the presence of the "fork and exec" mechanism. Note that at the time of our study, the new and popular Apache HTTP server [3] (derived from the NCSA HTTP server) was not widely available.

WebStone [62], developed by Silicon Graphics Incorporated, is designed to measure the performance of Web servers in multiple scenarios which reflect different Web site profiles. We have selected WebStone as the benchmark for our HTTP performance experiments because it is widely recognized to be the current industry standard for measuring Web server performance. WebStone performs a series of HTTP 1.0 [9] "GET" requests to a Web server and measures the server's connection rate, error rate, latency and throughput. The benchmark runs exclusively on clients and is independent of the Web server, its architecture, operating system and hardware. Thus, all measurements are taken from the client's side.

FTP Daemon and Client

We have selected the Washington University FTP daemon (wu-ftpd) [66] for the FTP server tests. It is a full-featured replacement for the standard FTP daemon supplied by most UNIX vendors supporting the FTP protocol defined in RFC 959 [52, 57]. Wu-ftpd is a popular FTP daemon used on many anonymous FTP sites around the world. Its source code is freely available, which allows us to port it to native ATM.

There is currently no accepted standard performance benchmark tool for FTP servers. We instead use the FreeBSD FTP client [39], which is not specifically written for benchmarking, but is freely available for our modification. The program allows the user to transfer files to and from a remote network site using the FTP protocol [52, 57] while recording FTP transaction times.

To convert the FreeBSD FTP client program for benchmarking purposes, we have modified it so that it can accept FTP commands from script files, which automate the login process and the file transfer.

2.5.3 Summary

In this chapter, we present background and research related to this thesis. The next chapter describes the architecture and network implementation of the benchmarks and applications described in this section. We also describe how they have been ported to the native ATM environment.

Chapter 3

Network Applications and Porting Them to Native ATM

Most of the network benchmarks and applications we use in our experiments do not originally support the native ATM network environment, and therefore require modifications. This chapter begins by describing the approach we use to port these applications to Fore System's native ATM API. The actual porting depends on each program's architecture, and is described in detail in later sections. For each application, we provide an overview of its networking architecture, compilation and setup, network input and output operations, timing (if any), and modifications made for porting to the native ATM environment.

3.1 General modification approach

This section explains the general approach we apply to port network applications which are originally implemented using the Sockets Application Programming Interface (or Sockets API) to the Fore System's API (or Fore API). The actual modifications differ among the applications themselves. The sockets and Fore ATM network implementations are described in Sections 2.2 and 2.3.3 respectively.

3.1.1 The Makefile and Header Files

In order to use the Fore API's function calls, modifications are required in the application's *Makefile*. Both server and client applications require the inclusion of the paths of the Fore Systems' directory (`$FORE_DIR`), Fore's ATM header files' directory (`-I$(FORE_DIR)/include`) and its library (`-L$(FORE_DIR)/lib -latm`). The socket library *socklib* (see Section 2.2.2) used for the Sun's is retained in some applications because it is required for other purposes in those applications (e.g., Netperf and Wu-ftpd both maintain an extra socket connection used for control during the entire session).

The following Fore ATM headers must also be included in the application's source code:

```
#include <fore/types.h>
#include <fore_atm/fore_atm_user.h>
```

3.1.2 The Socket Functions

The Fore API provides an abstraction with a style resembling that of the Sockets API. Taking advantage of this similarity, we attempt to replace each socket function with the corresponding Fore

API function, with the objective of keeping source code modifications to a minimum. Figure 3.1 compares the sequences of function calls in both Socket API and Fore API server implementations at the application level, while Figure 3.2 compares those in their client counterparts. The two implementations are explained in the rest of the section.

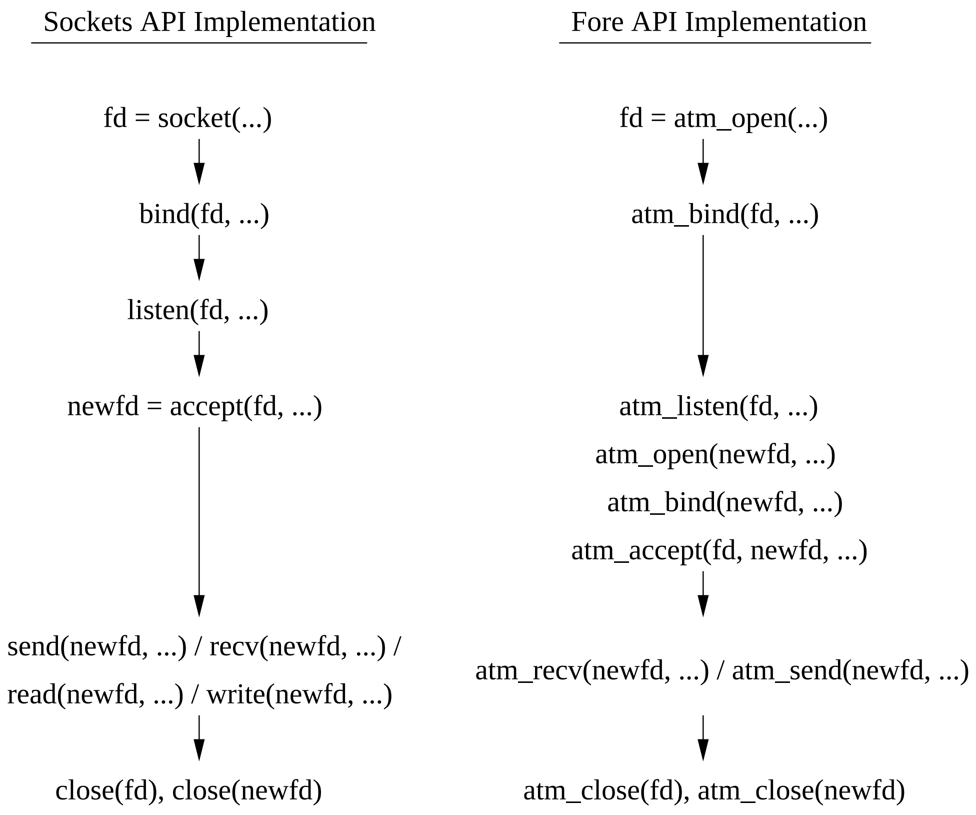


Figure 3.1: Socket API vs Fore API server implementations

Creating a connection end-point

The procedure used to create a connection end-point is similar for both client and server applications. Both Sockets API and Fore API represent a connection end-point in the form of a file descriptor (`fd` in Figures 3.1 and 3.2). To identify a connection end-point, the Sockets API uses a "socket" which is specified by a host and a port number, whereas the Fore API uses an "ATM end-point" which consists of a Network Service Access Point (NSAP) and an Application Service Access Point (ASAP).

The `socket()` function creates a socket and returns a file descriptor for the new socket. We replace this function with `atm_open()` which instead opens an ATM connection end-point. For example,

```
   fd = socket(AF_INET, SOCK_STREAM, IPPROTO_TCP);
```

becomes

```
   fd = atm_open(ATM_DEVICE, O_RDWR, &atm_info);
```

where `ATM_DEVICE` specifies the ATM device ("`/dev/fa0`" in our system) and `O_RDWR` specifies the read/write mode to pass to the `open()` system call. The `atm_open()` function returns the descriptor of the ATM end-point, and information related to the end-point is returned in `atm_info` (in the

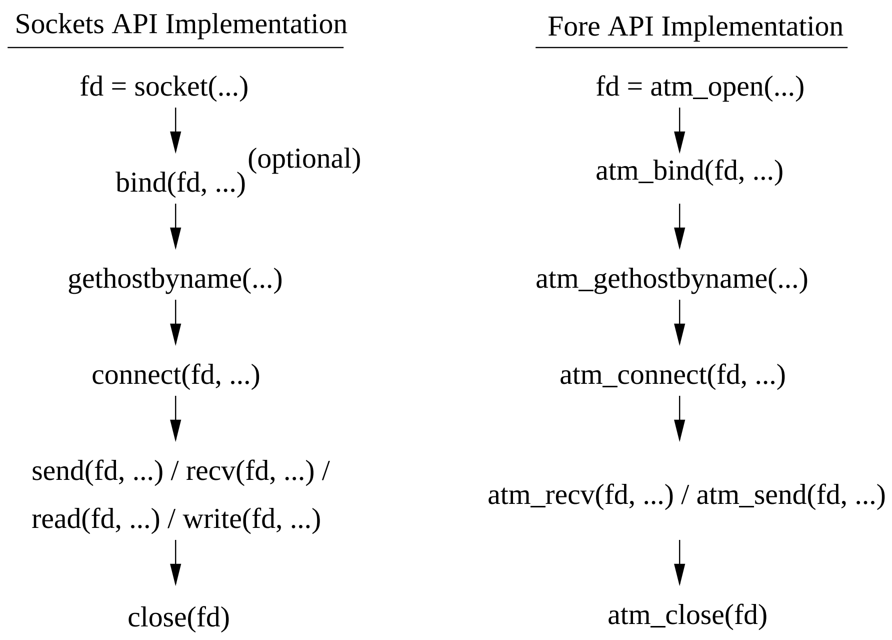


Figure 3.2: Socket API vs Fore API client implementations

current version of the Fore API, this structure only contains the ATM MTU size). Porting to the Fore API also requires changing the address structure of the server and client from the Sockets API type `sockaddr_in` to the Fore API `Atm_endpoint`. For example,

```
typedef struct sockaddr_in SERVER_SOCK_ADDR;
typedef struct sockaddr_in CLIENT_SOCK_ADDR;
```

becomes

```
typedef struct Atm_endpoint SERVER_SOCK_ADDR;
typedef struct Atm_endpoint CLIENT_SOCK_ADDR;
```

Binding

In a Sockets API server application, the function `bind()` binds a “name” to a new socket, containing its own network address and port number for incoming connections. This is normally not required in a Sockets API client application unless the application must connect with a specific port number. However, a client application using the Fore API is required to use `atm_bind()`. We replace the `bind()` function by `atm_bind()` which binds an ASAP with a new ATM end-point. For example,

```
bind(fd, name, sizeof(*name));
```

becomes

```
atm_bind(fd, asap, &asap, qlen);
```

The first `asap` is the ASAP proposed by the user. ASAP’s in the range of 0 through the defined constant `SAPS_RESERVED` (2047) are reserved for use by the ATM device driver. However, a value of 0 can be specified in `atm_bind()` to allow the ATM device driver to decide which ASAP to use. The second `asap` is filled in by the function call and is assigned the actual value used. This normally equals the user-proposed ASAP, unless that ASAP is already in use, in which case an unused ASAP is allocated. The `qlen` argument specifies whether the caller will be a server or a client — a value of zero indicates a client, while a non-zero value indicates that the caller will be a server, with the maximum number of outstanding connection requests limited by the value of qlen. We set this value to be the same as the backlog argument provided to the listen() call in the original Sockets API implementation,

```
listen(int fd, int backlog);
```

since both represent the maximum number of outstanding connections permitted.

The Netperf server (*netserver*) (see Section 3.2) sets its backlog to 5 for the control connection and 1 for the data connection (this is reasonable since there is always one data connection per client). The TTCP's receiver (Section 3.3) and the FreeBSD FTP client (the server side of the data connection, see Section 3.7) also use backlog values of 1 for the same reason. The NCSA HTTP daemon (Section 3.4) sets the value to 35, because many more connection requests are expected for a normal HTTP server. The Wu-ftpd FTP daemon (Section 3.6) sets the value to 1 because all additional connection requests are handled by the Internet Daemon program (inetd).

Listening and accepting a connection on a server

The function atm_listen() blocks until a pending connection is present, whereas in the Sockets API implementation, the blocking occurs not in listen() but in accept() (unless the socket is set to be non-blocking [58]). Unlike accept() which automatically creates and binds a socket for the new connection, atm_accept() requires a new ATM end-point to be created and bound explicitly beforehand.

In our modified versions of the applications, the "listen" function is postponed until a later stage (before the Fore API's "accept" function) , as we want the blocking to occur at the same place in both implementations. As a consequence, the call

```
accept(fd, &client, &len);
```

is replaced by

```
atm_listen(fd, &conn_id, &atm_client, &qos, &aal);
atm_accept(fd, newfd, conn_id, &qos, duplex);
```

The last four arguments of atm_listen() are filled upon the return of the call: conn_id contains a connection identifier to be used with atm_accept(), qos specifies the client's requested quality of service (QoS) and aal specifies the client's requested ATM Adaptation Layer to be used (the current Fore API version only supports AAL3/4 and AAL5, and AAL5 is used in our experiments). A new ATM end-point must be created using atm_open() and bound using atm_bind() before it is referenced by atm_accept() (shown in Figure 3.1 as newfd).

Client's connection

A client is usually given only the host name of the server, and then the function gethostbyname() is called to obtain the server's IP address. This is followed by a call to the function connect() to connect to the particular socket on the server host. The Fore API version uses atm_gethostbyname() to obtain a server's ATM address (internally, atm_gethostbyname() finds the ATM address by first calling gethostbyname() to determine the IP address, and then looking for that address in the ATM device driver's ARP table). For example, we replace

```
hostent = gethostbyname(hostname);
```

by

```
atm_gethostbyname(hostname, &addr);
```

After the atm_gethostbyname() call, the addr argument will contain the ATM address of the server host.

We also replace connect() with atm_connect():

```
   connect(fd, &server, sizeof(server));
```
becomes
```
   atm_connect(fd, &addr, &qos_req, &qos_sel, aal_type_5, duplex);
```

The `qos_req` argument specifies the caller's requested network bandwidth requirements, and `qos_sel` is used to return the bandwidth resources actually allocated (the requested values may be adjusted downward by the device driver).

Network Input and Output

The `read()` and `write()` functions are replaced by `atm_recv()` and `atm_send()` respectively, with the function arguments preserved. As described in the next few sections, some applications use other I/O functions which we handle specifically.

Termination of connection end-point

Sockets are terminated using the `close()` function. Similarly, ATM end-points are terminated using `atm_close()`. We therefore replace every occurrence of `close()` for the particular socket by `atm_close()` in the Fore API implementation.

3.1.3 Reliability and Flow Control

The native ATM network leaves the responsibility of a reliable data transfer to the application layer. The consequence is that native ATM applications may require some sort of reliability scheme. We validate data transfers (i.e., the HTTP and FTP files) but not the HTTP requests/responses and FTP commands/replies, since they are automatically validated by the applications.

In designing a suitable reliability scheme for our network applications, we are concerned with the following issues:

- AAL5 performs the CPCS-PDU level checksum in which corrupted CPCS-PDU's are dropped. In other words, all CPCS-PDU's that arrive at the receiver side will contain no corrupted data. Therefore, we know that the entire data unit is correct if the expected number of bytes are received.
- In the thesis we are concerned with performance issues in local area networks, and as a result are assuming that data loss will occur infrequently.
- We wish to implement a simple algorithm which can achieve the task adequately without greatly sacrificing performance. An intensive performance study of reliable and flow control algorithms is beyond the scope of this thesis.
- A major cause of the ATM cell drops is the overflow of memory buffers (e.g., ATM switch buffers and kernel buffers). One way to minimize any kind of overflow is to apply a flow-control algorithm.
- Since overflows cannot be completely prevented, some kind of retransmission scheme is required.

Figure 3.3 illustrates the "reliable flow-control algorithm" we implement for the network applications, which is a simple algorithm derived from the window-based TCP protocol [50]. The server sends the data in consecutive "sessions", each beginning with a message informing the receiver of the number of bytes to receive and ending with an acknowledgment from the receiver. Figure 3.4 shows an example with three sessions. We call the number of bytes sent during each session the "ATM window size" (or w). The sender begins each session by sending the integer w, which is also the number of bytes the other end is expected to receive during that session (w is "40" in Figure 3.4). The sender then sends this number of bytes in data units with size equal to the data buffer size b (10 bytes in the example), and waits for the receiver to acknowledge receiving the w bytes. The receiver returns an acknowledgment which equals the number of bytes it has received during the session ("40" in the example). If no data is lost, upon receiving the acknowledgment, the sender starts another session. This is repeated until all the data is sent.

Sender:

```
b = buffer size
w = original window size
t = file size (total bytes to send)
while $t > 0$ (there are bytes to send)
    $w = min(w, t)$
    $ack = w$
    send $ack$ to receiver
    while $ack > 0$
        $n = min(ack, b)$
        read $n$ bytes from file, send to receiver
        $ack = ack - n$
        $t = t - n$
    receive $ack$ from receiver
    if $ack = -1$ (retransmission required)
        rollback file pointer
        $t = t + w$
        $w = w/2$ (adjust window size)
        adjust $w$ to next lowest multiple of $b$
    $ack = -1$ (transmission ends)
    send ack
```

Receiver:

```
$r$ : total bytes to receive in current session
while $r \neq -1$
    $ack = 0$ (bytes currently received)
    receive $r$ from sender
    while $r > 0$ and time not expired
        start timer
        if time expired
            roll back file pointer
            $ack = -1$
        else (time not expired)
            receive data from sender
            $n$ = number of bytes received
            write data to file
            $r = r - n$
            $ack = ack + n$
    send $ack$ to sender
```

Figure 3.3: The reliable flow-control algorithm

If an ATM cell happens to be dropped, the complete CPCS-PDU is discarded, as is the data unit containing this data ("G" in the example). If the expected number of bytes is not received, the receiver waits for a specified period of time in case the data is in the process of transmission. If no additional data is received within this time period, it returns a negative acknowledgment (represented by a "-1"). Upon receiving the negative acknowledgment, the sender halves the ATM window size (w becomes "20" in the example) and retransmits the data. This is an attempt to reduce the chance of further data loss in the network. The new ATM window size is adopted and will be halved again whenever a retransmission occurs (i.e., it is never increased). A pre-defined ATM window size is always used at the beginning of each transaction.

To add the reliable flow-control mechanism to a native ATM application, we replace the loops responsible for the network input (containing `atm_recv()`) and output (containing `atm_send()`) with two new functions which implement the algorithm in Figure 3.3. The initial ATM window size is also specified.

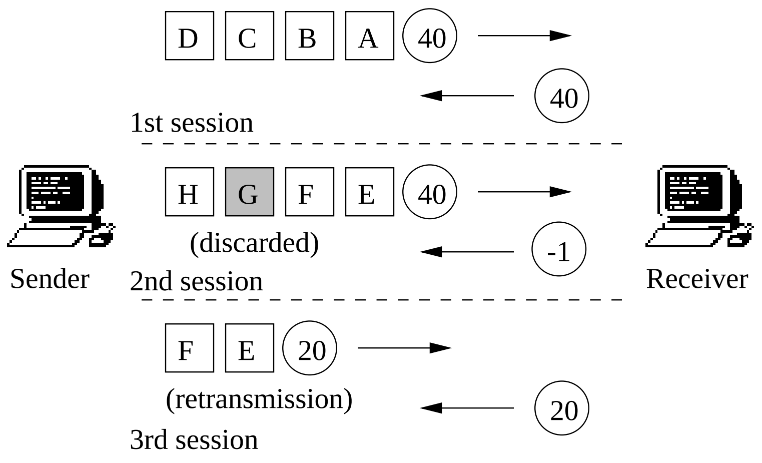


Figure 3.4: Example of the reliable flow-control algorithm

3.2 Netperf

Netperf [21], developed by Hewlett-Packard, is a networking performance tool widely used in the computer industry as a standard benchmarking tool. It is capable of performing "memory-to-memory" transfer tests on a variety of platforms. It supports a variety of API's and protocols including the Sockets API and the Fore ATM API. Netperf version 2.1(pl3) is used in our experiments.

3.2.1 Program Architecture

Netperf consists of two executables: the server program *netserver* runs on the server host and the client program *netperf* on the client host. The *netserver* program initially listens on a TCP port, and *netperf* connects to this port to establish a "control connection". Information regarding test configurations is sent over the control connection to the *netserver* program, which responds with its own network information and opens a "data connection". Regardless of the type of test being run, the control connection is always a TCP connection using the Sockets API, whereas the data connection can be any API or protocol supported by Netperf (e.g., Sockets API, XTI or the Fore API). Netperf places no additional traffic on the control connection while a test is in progress [21].

The *netperf* client connects to the data connection and sends streams of data for a specified period of time (alternatively, one can specify the amount of data to be sent, but this is only supported in the Sockets API test and not the Fore API test). The data being sent consists of random byte patterns in order to avoid data compression performed by any part of the network during the transmission, if any. The *netserver* program receives the data, and when the transfer is completed, sends its statistics to *netperf* over the control connection. It then closes both connections and *netperf* processes the statistics and prints the results.

3.2.2 Compilation, Setup and Execution

The Fore API is supported by Netperf but is disabled in the original compilation settings. This feature is enabled before compiling these programs in our testbed. The executables *netserver* and *netperf* are created on both the Sun's and SGI's without any problem.

Before executing *netperf*, *netserver* must be running on the server host and listening to a specified TCP port. The user starts *netperf* by providing configuration information (in the form of command-line arguments) such as the type of test (e.g., bulk data transfer or request/response), the type of data stream (e.g., TCP, UDP or ATM), the duration of the test, and the data and socket buffer sizes to be used. Default values are used for those parameters not specified.

3.2.3 Network Input and Output

The initial exchange of network test configurations and network information between *netperf* and *netserver* are performed using the `send()` and `recv()` system calls. The *netperf* client sends data using the `send()` call for the Sockets API and `atm_send()` call for the Fore API. The *netserver* program receives data using `recv()` for the Sockets API and `atm_recv()` for the FORE API. The user is allowed to adjust the data buffer sizes, as well as the send and receive socket buffer sizes on the client and server if desired. If not specified, the data buffer size used by *netperf* is the same as its send socket buffer size, and the data buffer size used by *netserver* is the same as its receive socket buffer size. The default send and receive socket buffer sizes under IRIX 6.2 are 60 KB. Under Solaris, the default socket buffer sizes are not properly returned by the appropriate system call, and Netperf assigns the value of 4 KB to each.

3.2.4 Timing

The current version of Netperf only provides timing capabilities on the client side (the sender). The function `gettimeofday()` is called to record the start time after the client issues the `connect()` call for the data connection and before any data is sent over this connection. The function `shutdown()` is issued after all data is sent, followed by a `read()` system call. Since the receiver will not send any additional data, this call will block until the receiver closes the connection. The `gettimeofday()` function is then called a second time and the elapsed time is calculated. This method ensures that the timer stops after all data has been received by the other end.

3.2.5 Modifications for Native ATM

The version of Netperf used is capable of supporting the Fore API, and thus no modifications are required.

3.3 TTCP

Test TCP (TTCP) [64], created at the US Army Ballistic Research Lab (ARL), is designed for testing the overall performance of bulk data transfers using TCP or UDP between two systems on IP-based networks. TTCP is capable of performing "memory-to-memory transfer" tests (i.e., the transferred data is resided in memory) as well as "file transfer" tests (i.e., the transferred data is from a file). TTCP version 1.0 is used in our experiments.

3.3.1 Program Architecture

TTCP evaluates the data transfer rate from a "sender" to a "receiver". Unlike Netperf, TTCP uses the same program with different modes for sending and receiving. A "transmitter" mode is used for the sending host and "receiver" mode is used for the receiving host. The receiver must be running before the sender starts. The data to be transferred can be taken from standard input or a specified file. Alternatively, the sender can create ("source") a random data pattern and the receiver can discard ("sink") the data if desired. The data is sent using a single buffer, with the size specified by the "buffer length" (equivalent to the data buffer size). If the data is from a file, the entire file is sent. Otherwise, the data size can be determined by the user, by specifying the number of times the buffer is sent. The default is 2048 times.

3.3.2 Compilation, Setup and Execution

An executable *ttcp* has been created by compiling a single source file. A minor modification to the argument types in the function `select()` is required to resolve an incompatibility problem with both the SGI and Sun platforms:

```
(void)select(1, (char *)0, (char *)0, (char *)0, &tv);
```

has been changed to

```
(void)select(1, (fd_set *)0, (fd_set *)0, (fd_set *)0, &tv);
```

Various options such as the socket buffer size and port number can be specified in the form of command-line arguments.

3.3.3 Network Input and Output

In TCP mode, TTCP invokes the `write()` system call to send data and `read()` to receive data. The data buffer sizes are specified by the user and are passed as arguments to `write()` and `read()`. TTCP uses default data buffer sizes of 8 KB for reading and writing data.

3.3.4 Timing

TTCP provides timing capabilities on both the transmitter (client) side and receiver (server) side. On the transmitter side, the function `gettimeofday()` is called to record the start time after the `connect()` socket call and before sending any data. It is called again to record the stop time after all data is sent. On the receiver side, `gettimeofday()` is called after the `accept()` socket call and before receiving any data. It is called again to record the stop time after all data is received. Note that the sender does not wait for all the data to arrive on the other side. Hence, it is possible for the sender to have a higher throughput than the receiver, due to the presence of network delay.

3.3.5 Modifications for Native ATM

We have modified TTCP to support the Fore API using the approach described in Section 3.1. An ATM command-line flag is added to use the ATM transfer mode. For the reliable ATM version, an extra command-line flag is added to specify the ATM window size. When no ATM flags are specified, the original Sockets API implementation is used. Hence, the `socklib` library is retained for the Sun's.

3.4 NCSA HTTP Daemon

The NCSA HTTP daemon (NCSA HTTPd) [36] is an HTTP 1.0 [9] compatible server developed at the National Center for Supercomputing Applications at the University of Illinois. NCSA HTTPd version 1.5.2a, the latest available at the time, was chosen for our experiments.

3.4.1 Program Architecture

From version 1.4 onwards, the NCSA HTTPd supports "pre-forking" [36]. In traditional client/server architectures, the master process on the server side accepts requests from the client side and then spawns a new process to handle each new request. The time and resources required by the `fork()` and `exec()` operations are significant, particularly in light of the fact that a typical HTTP request is very short. Pre-forking reduces the overhead required to handle requests by spawning a number of children before receiving any requests. In other words, the children are spawned while the main process is waiting for requests. Figure 3.5 shows the steps taken by the NCSA HTTPd for handling requests:

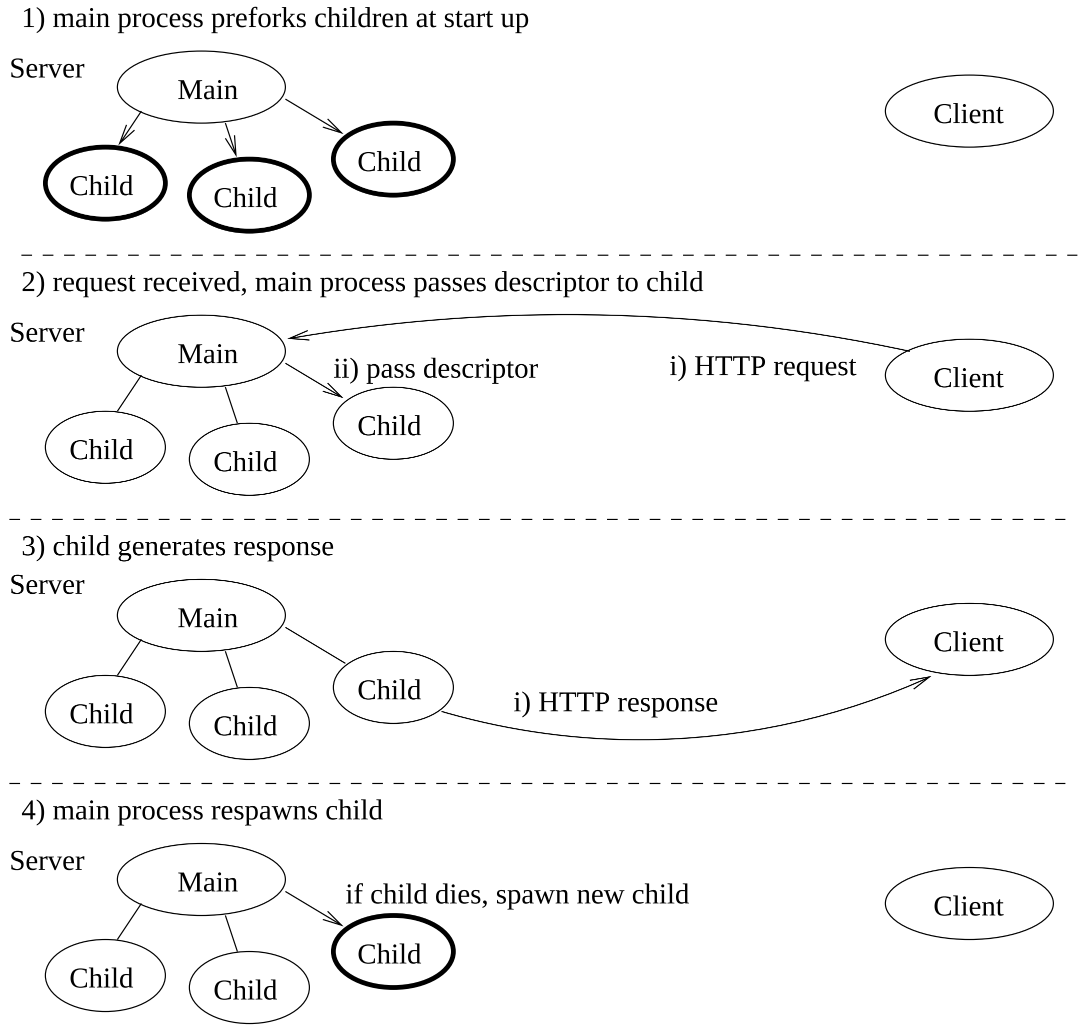

1) main process preforks children at start up
Server
Main
Child
Child
Child
Client
2) request received, main process passes descriptor to child
Server
Main
ii) pass descriptor
i) HTTP request
Child
Child
Child
Client
3) child generates response
Server
Main
Child
Child
Child
i) HTTP response
Client
4) main process respawns child
Server
Main
if child dies, spawn new child
Child
Child
Child
Client

Figure 3.5: Pre-forking in NCSA HTTPd

1. When the HTTP daemon is started, the main process spawns a pool of processes and then listens for incoming requests (three child processes are shown in Figure 3.5). The number of processes in the pool can be specified by the user. The default is 5.

2. Upon receiving a request, the main process distributes it (in the form of a socket descriptor) to the first idle child process in the pool. If there are no free processes, a new one is created.

3. After the child completes its task, it goes back to the idle state and waits for a new request from the main process. If it has already handled a pre-specified number of requests, it notifies the main process and dies. This design allows some systems to avoid problems after prolonged use, such as memory leaks. The default number of requests per child is 100.

4. If the child dies, the main process spawns another child to replace it. Thus, it always tries to keep a fixed number of child processes in the pool.

3.4.2 Compilation, Setup and Execution

The standard NCSA HTTPd distribution was compiled on both the Sun's and SGI's without any incompatibilities. The server consists of an executable program *httpd* which is configured at start up time using configuration files that specify the necessary parameters such as the server's root directory, default port number and initial number of pre-forked child processes. The server runs as a background process.

3.4.3 Network Input and Output

The NCSA HTTPd receives HTTP requests from the clients using the `read()` system call, which reads the network input data line by line with a data buffer size of 8 KB. Sending an HTTP response involves issuing a combination of four functions of the NCSA HTTPd: the function `send_fp()` writes a directory list to the socket using the system call `write()` with a data buffer size of 8 KB; the function `rprintf()` writes a formatted line to the socket using the `vfprintf()` function; the functions `rputs()` writes a string to the socket using `fputs()`; and the functions `rputc()` writes a single character to the socket using `putc()`.

3.4.4 Modifications for Native ATM

To port NCSA HTTPd to the Fore API, we have modified the corresponding socket functions using the approach described in Section 3.1 with the following additional modifications:

- The original default TCP port is "80". In the Fore implementation, ASAP's in the range of 0 through the defined constant `SAPS_RESERVED` (2047) are reserved for use by the ATM device driver. Therefore, we arbitrarily assign the value "8000" to be the default ASAP.

- The functions `which_host_conf()` and `rfc931()`, which require structures of the Sockets API type `sockaddr_in`, are by-passed in our ATM implementation. This should not affect the program in any way because the function `rfc931()` (for authentication server [56]) is not used by default in the Sockets API implementation. The function `which_host_conf()` is used to call `gethostbyname()` to determine the name of the local host. In the Fore API implementation, this is done by using the `atm_gethostbyname()` function.

- The NCSA HTTPd's function `rprintf()` uses the `vfprintf()` function to write a formatted line to the socket, but the Fore API has no equivalent function. Replacement of this function requires an additional step: the function `vsprintf()` is called to produce the formatted string which is stored in a temporary buffer and is then sent using `atm_send()`:

  ```
  vfprintf(socket, string, argList);
  ```

  becomes

```
char atmbuf[MAX_STRING_LEN];
n = vsprintf(atmbuf, string, argList);
atm_send(atm_endpoint, atmbuf, n);
```

3.5 WebStone

WebStone [62], developed by Silicon Graphics Incorporated, is designed to measure the performance of Web servers under multiple scenarios, using metrics such as connection rate, error rate, latency and throughput. WebStone version 2.0.1 is used in our experiments.

3.5.1 Program Architecture

WebStone runs a master program on one host to control one or more copies of a slave program on other hosts (the master and slave programs may run on the same host). The slave program, called the "load generator", emits a stream of HTTP requests (known as the "workload") and measures the system response. The master program starts by issuing a "go" message to all the slaves. The slaves then generate requests chosen randomly from a user-specified list, and send them to the HTTP server. After a specified amount of time, the master issues a "stop" message to the slaves and statistics are returned back to the master. Figure 3.6 illustrates this architecture, showing an example of a running WebStone with three slaves.

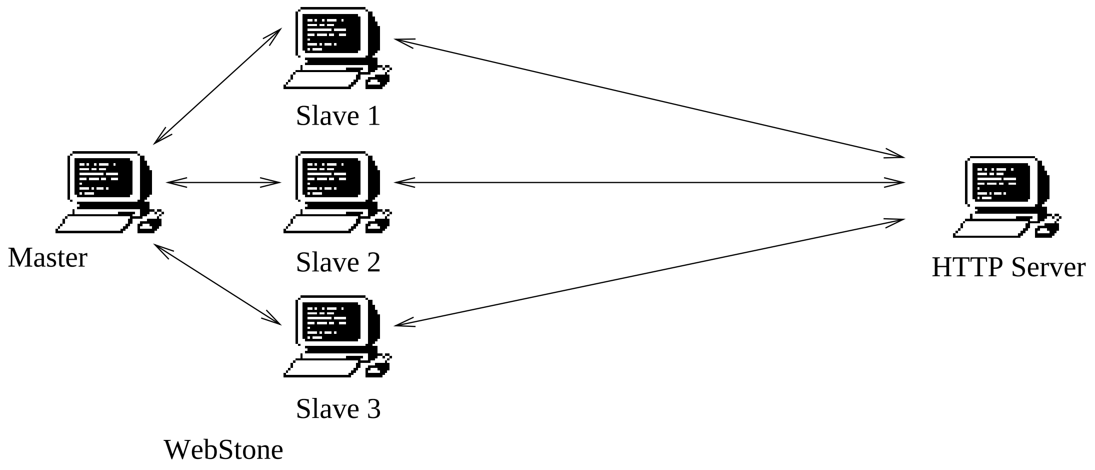


Figure 3.6: WebStone architecture

3.5.2 Compilation, Setup and Execution

Since WebStone was specifically written for the SGI's, installation is straightforward on that platform. Installing WebStone on the Sun's requires a few modifications. In particular,

- The `#error` pre-processor constructs used for comments are supported on the SGI's but not the Sun's, and have been safely removed;
- On the SGI's, the signal handling function may use an integer argument or no argument:

  `void (*signal (int sig, void (*func)(int)))(int);`

or

`void (*signal (int sig, void (*func)()))();`

On the Sun's, only the former is accepted. So we have inserted a dummy integer argument in the declarations of the corresponding functions. For example,

`void alarmhandler(void);`

becomes

`void alarmhandler(int dummy);`

- the WebStone script *runbench* uses the environment variable `localhost` to determine the local host name of a machine. This is not defined by default in our systems. We use the UNIX program *hostname* to achieve the same goal;
- WebStone is supposed to allow users to define the HTTP server's port. In the original code, however, the port number is fixed to the default "80". We have modified the code to allow the user to successfully specify the port number using the configuration file.

WebStone refers to two files for user-defined parameters: the configuration file, containing information such as the names of the server and client hosts, and the file set, containing the names of files to be requested from the server. WebStone also provides a utility program that generates files of particular sizes for retrieval.

WebStone consists of several scripts and programs. At the top level, the user starts by running the *webstone* script on the master host on the client side, which calls another script *runbench.* Then *runbench* reads the test configuration files and distributes them to the slave hosts along with the executable program *webclient.* The master program *webmaster* is started on the master host which signals the *webclient* programs to start on all slave machines. The HTTP daemon must be running on the server host during the tests. When the tests are completed by all slaves, results are collected and a text file is produced containing a single summary report.

3.5.3 Network Input and Output

WebStone creates a request string and invokes the `write()` system call to send each HTTP request to the server. A small 256-byte data buffer is used because HTTP requests are usually small.

To receive the HTTP response, WebStone (*webclient*) fetches the HTTP header and body by invoking the `read()` system call. The function is called more than one time if the data size is larger than the data buffer size. Since HTTP responses are usually large, an 8 KB data buffer is used.

3.5.4 Timing

All times are measured by the slaves. Each slave issues the `gettimeofday()` function to obtain time stamps at different points during an HTTP transaction. We are interested in the "client throughput". This is calculated based on the transfer time, which is the time difference between the time stamp before the `socket()` call and the time stamp after the `close()` call of the HTTP connection.

3.5.5 Modifications for Native ATM

Two new command-line flags are added to the main *webstone* script for switching to native ATM and the reliable native ATM versions of *runbench*, *webmaster* and *webclient*. The original Sockets API versions are executed if no flag is specified.

We only modify the connection between the WebStone clients and the HTTP server to support native ATM. Communication between the WebStone client and master still uses TCP connections. Since these connections are only used for passing information related to the results of experiments, their use has no effect on the measured throughputs. Webstone defines two functions for creating socket connections: the `passivesock()` function is used by the *webmaster* to connect to the *webclient*, and the `connectsock()` function is used by the *webclient* to connect both to the *webmaster* and the HTTP server. Since establishing the TCP connections between the *webclient* and *webserver* also involves `connectsock()`, we cannot modify this function directly. In our implementation, we introduce a new function specifically for the ATM connection. Other socket functions are directly replaced by the corresponding Fore API functions as described in Section 3.1.

3.6 Washington University FTP Daemon

The Washington University FTP daemon (Wu-ftpd) [66] is a full-featured replacement for the standard system FTP daemon supplied by most Unix vendors. The most recent version, Wu-ftpd version 2.4.2 (beta 15), is used in our experiments.

3.6.1 Program Architecture

Communication between an FTP client and the Wu-ftpd FTP daemon requires the use of the Internet Daemon (inetd) on the server which listens on a number of ports for requests for services. It reserves two TCP ports for the FTP daemon, one for a control connection and another for a data connection. As soon as inetd receives a connection request on the control port, it starts Wu-ftpd to handle the client's request. Both control and data connections are then established directly between Wu-ftpd and the client. The control connection lasts for the entire FTP session, but one data connection is used for each data transaction. Figure 3.7 depicts the structure of these connections.

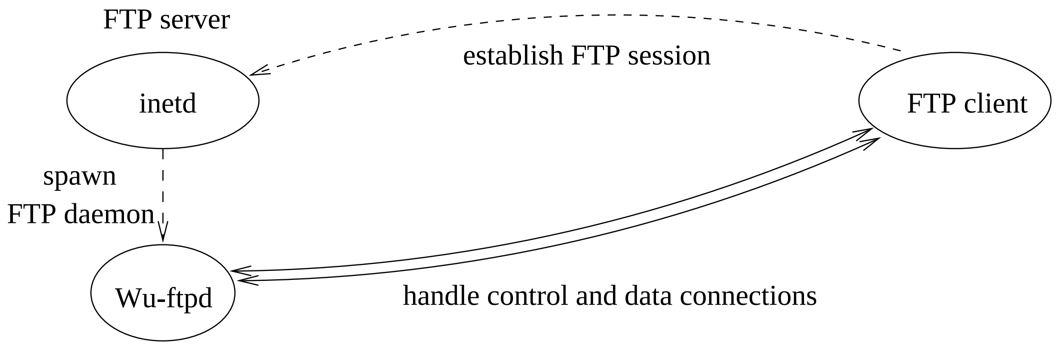


Figure 3.7: Wu-ftpd via the Internet Daemon

3.6.2 Compilation, Setup and Execution

No modifications are required for either the SGI's or the Sun's. The location of several configuration files is specified prior to the compilation of the Wu-ftpd program. The compiled Wu-ftpd is a single executable called *ftpd*. Two lines are added to the system's */etc/services* file to accommodate the two new TCP ports used by the FTP daemon (so as not to interfere with regular FTP services).

3.6.3 Network Input and Output

Wu-ftpd uses `getc()` to receive the client's requests and a combination of `printf()` and `putc()` to reply over the control connection. Three functions of Wu-ftpd are used to handle network input and output operations over the data connection:

- `send_file_list()` sends a directory list to the client. It reads a directory and sends its files and their information line by line in a formatted string using `fprintf()`;
- `send_data()` sends data to the client. This is used when a client performs a "get" operation. In ASCII mode, the function reads input from the specific file one character at a time and sends it using the function `putc()` one character at a time. In BINARY mode, it reads input from the file one data buffer at a time and sends it using the `write()` system call one data buffer at a time. The data buffer size used under both platforms is set to `BUFSIZ` defined in `<stdio.h>`, which is 4 KB under IRIX and 1 KB under Solaris;
- `receive_data()` receives data from the client. This is used when a client performs a "put" operation. In ASCII mode, the function reads data one character at a time using the function `getc()`, and writes data to the specified file one character at a time. In BINARY mode, it reads data from the specific socket using the system call `read()` with a data buffer size of `BUFSIZ` (4 KB in IRIX and 1 KB in Solaris), and stores the data in the destination file by writing the buffer using the `write()` system call.

3.6.4 Modifications for Native ATM

We port only the data connection to the Fore API and leave the control connection unchanged. This does not affect our experimental results because we only measure transactions using the data connections. Modifications to the data connection functions generally follow the approach described in Section 3.1.

The FTP specification requires the data port number of the FTP server to be one less than its control port number [52], and the default data port is "20". Again, because Fore ASAP's less than 2048 are reserved for the ATM device, we arbitrarily assign ASAP "4096" for the use with data port. This reassignment is feasible because under the FTP protocol [52] (see Section 2.5.1), the FTP client is the server side of the data connection, therefore does not require the FTP server to have a fixed data port number. On the contrary, the FTP server must know the client's data port number in order to make a connection. As described in Section 3.7.5, we use the FTP "port" command on the client to inform the server about the its data port number. Since the arguments of the "port" command will contain the client's NSAP as well as it's ASAP, additional modifications are made to Wu-ftpd to extract and validate these numbers.

3.7 FreeBSD FTP

Since there are no FTP benchmark programs that we are aware of, we use the FreeBSD FTP program [39] to measure FTP throughputs. The program provides facilities to transfer files to and from a remote network site using the FTP protocol [52, 57]. FreeBSD FTP version 5.28 is used in our experiments.

3.7.1 Program Architecture

The FreeBSD FTP program first requires the user to login to the FTP server using the Telnet protocol [51, 57]. It waits for the user to issue commands for file transfer and then sets up data connections and transfers data accordingly. The receiver's host name and port number can be adjusted using the FTP "port" command during an FTP session. There is also a built-in timer to measure the transfer time of each transaction.

3.7.2 Compilation, Setup and Execution

The FreeBSD FTP program was originally designed for BSD systems, thus a number of modifications are required in order to run it under IRIX and Solaris. Several modifications are necessary to resolve several type incompatibilities, header inclusions and signal handling functions. In particular:

- the use of the *gcc* compiler on the Sun's has reduced a number of type incompatibilities. For example, `signal()` and `setsockopt()` are of type

  `void (*signal (int sig, void (*func)(int)))(int);`,

  and this is allowed by *gcc* but not by the original *cc* compiler;

- some constants used by FreeBSD FTP are not originally defined on the two platforms and are required to be defined explicitly. They are `L_INCR` and `ECHO` on the Sun's and `NCARGS` on the SGI's;

- some additional header files are included: `<sgtty.h>`, `<dirent.h>` for the Sun's and `<sys/ttold.h>` for the SGI's.

To convert the FreeBSD FTP program for benchmarking purposes, we have modified it so that it can accept FTP commands from script files. We have modified its transaction report messages to include additional information about the sizes of various data buffers used. The standalone executable *ftp* is created during compilation.

3.7.3 Network Input and Output

The FreeBSD FTP program issues commands by forming a formatted string and sending it over the control connection using the function `vfprintf()`.

When sending data over the data connection in BINARY mode, a specific file is read with a data buffer of size `BUFSIZ` (4 KB on the SGI's and 1 KB on the Sun's) and is sent with that buffer using the `write()` system call. In ASCII mode, the data is read and sent one character at a time.

When receiving data over the data connection in BINARY mode, data from the network is read using the `read()` system call, placed in an intermediate data buffer, and then stored in the destinated file. The size of the data buffer used is the system's preferred I/O block size specified by `st_blksize`, which is 32 KB on the SGI's and 8 KB on the Sun's. In ASCII mode, the data is read and stored one character at a time.

The mismatch between the data buffer sizes used by the FreeBSD client and the Wu-ftpd server results in very different throughputs in the two transfer directions obtained in our FTP transfer experiments (see Section 4.6). Note that the FreeBSD client also uses different data buffer sizes in the two transfer directions.

3.7.4 Timing

The original FreeBSD FTP program only records the time of the data transfer. The function `gettimeofday()` is called before any data is read from the file (for sending) or from the network (for reading). It is called again immediately after all the data is sent to the network (for sending) or stored on the disk (for reading). The elapsed time is then calculated.

As we have seen in Figures 2.9 and 2.10, the sending time may be faster than the receiving time. To measure the FTP transaction time in a more accurate way, we modify the program such that when it is the sender (i.e., when the "put" command is used), it waits for all the data to arrive at the server and the connection to close before calling `gettimeofday()` the second time. This is the same method applied by Netperf with the use of the `shutdown()` call (see Section 3.2.4). We also modify the program so that the measured time spans the entire data connection — the first `gettimeofday()` is called just before the `socket()` call, and the second is called after the `close()` call on the connection.

3.7.5 Modifications for Native ATM

In our native ATM implementation, the control connection, using the Sockets API over TCP, is unaltered (see Section 3.6). Modifications to the data connection functions generally follow the approach described in Section 3.1.

The FTP protocol specifies the same default port for the client's data and control connections [52, 57]. This is not possible in our native ATM port since the control connection is a TCP connection. We resolve this problem by allowing the client to send its NSAP and ASAP to the server via the control connection, so that the server can establish the data connection to the client. To achieve this, the "port" FTP command is modified to send the NSAP and the ASAP to the server. Modifications are also made to the "port" command and the composition of its arguments (the NSAP and ASAP). It originally requires two integers to represent the TCP port number (a socket port is implemented as an unsigned short integer). In the native ATM version, it requires four integers for the NSAP and another four for the ASAP (each implemented as an unsigned integer).

We ensure that the "sendport" flag is switched on by default (this is not the case when the program is compiled on the SGI platform), so that the "port" command is issued at the beginning of every data transfer (this is not included in timing the transfer). The Fore API ASAP's in the range of 0 through the defined constant `SAPS_RESERVED` (2047) are reserved for use by the ATM device driver, and therefore we arbitrarily choose the ASAP "2100" for the client's data port.

Since the FTP experiments are only performed using BINARY mode, we do not modify the code responsible for ASCII mode transfers. The remaining Sockets API modifications follow the approach described in Section 3.1.

3.8 Summary of Modifications

Table 3.1 summarizes the default data buffer sizes used in the benchmarks and applications modified. The second column specifies the type of data transfer, the third column shows the function being used, and the fourth column indicates the buffer sizes used on the two platforms. Note that some function calls, such as `printf()` and `vprintf()`, do not require the data buffer size to be specified.

Benchmark/ Application	Type of data transfer	Function call	Data buffer size (bytes) Sun	SGI
Netperf	write (*netperf*)	`send()`	4096	61440
	read (*netserver*)	`recv()`	4096	61440
TTCP	write (*transmitter*)	`write()`	8192	8192
	read (*receiver*)	`read()`	8192	8192
NCSA HTTPd	HTTP request	`read()`	8192	8192
	HTTP response	`write()`	8192	8192
		`vfprintf()`	N/A	N/A
		`fputs()`	N/A	N/A
		`putc()`	1	1
WebStone	HTTP request	`write()`	256	256
	HTTP response	`read()`	8192	8192
Wu-ftpd	FTP request	`getc()`	1	1
	FTP response	`printf()`, `putc()`	N/A	N/A
	FTP send (binary)	`write()`	1024	4096
	FTP receive (binary)	`read()`	1024	4096
	FTP send (ASCII)	`putc()`	1	1
	FTP receive (ASCII)	`getc()`	1	1
FreeBSD ftp	FTP request	`vfprintf()`	N/A	N/A
	FTP response	`getc()`	1	1
	FTP send (binary)	`write()`	1024	4096
	FTP receive (binary)	`read()`	8192	32768
	FTP send (ASCII)	`putc()`	1	1
	FTP receive (ASCII)	`getc()`	1	1

Table 3.1: Summary of default data buffer sizes

Table 3.2 summarizes the modifications made to each of the programs. The second column indicates the total number of lines of the program's source code, including all necessary *Makefiles* and scripts. The third and fourth column list the number of lines modified or added for the unreliable and reliable native ATM versions respectively.

Benchmark/ Application	Total lines of code (compiled version)	Lines of code modified/added (unreliable ATM)	(reliable ATM)
Netperf	42657	0	0
TTCP	841	139	303
WebStone	5974	210	285
NCSA HTTPd	20684	65	120
Wu-ftpd	12330	358	485
FreeBSD ftp	5954	103	241

Table 3.2: Summary of modifications of the network benchmarks/applications

Chapter 4

Experiments

This chapter describes the experiments we conduct to evaluate the performance of the network oriented applications discussed in Chapter 3 when executing using the Ethernet, TCP/ATM and native ATM network environments. We begin with a description of our testbed, which is followed by a detailed discussion of the experiments and their results.

4.1 The York University LCSR Testbed

The Laboratory for Computer Systems Research (LCSR) at York University, in Toronto, Ontario, maintains an ATM testbed that is used to perform measurement studies and conduct a variety of network experiments. The hardware and software configurations of the testbed are introduced below.

4.1.1 Hardware and Software Configuration

The portions of the LCSR used for our experiments consist of the following hardware facilities:

- Four Sun SparcStation5 workstations, each with a 70 MHz MicroSparc II processor, 32 MB RAM, 16 KB instruction cache, 8 KB data cache, and no secondary cache.
- Four SGI Indy workstations, each with a 175 MHz R4400 processor, 64 MB RAM, 16 KB instruction cache, 16 KB data cache, and a 1 MB secondary unified instruction/data cache.
- A Cabletron MRXI-2 Ethernet hub.
- A Fore Systems ASX-200BX ATM switch [24], with 12 CAT-5 UTP ports.
- Fore Systems GIA-200E adapters [25] for the SGI's and SBA-200E adapters [22] for the Sun's.
- The SGI's use a Seagate ST31230N local hard disk with 1 GB storage capacity. It has a rotation speed of 5144 RPM, 512 KB cache, and uses a SCSI interface.
- The Sun's use a Conner CP30548 (also known as SUN0535) local hard disk with 535 MB storage capacity. It has rotation speed of 5400 RPM and uses a SCSI interface.

The Sun SparcStation5's are running Solaris 2.5.1 and the SGI's are running IRIX 6.2. The ATM switch software is ForeThought version 4.1.0.

4.1.2 Network Configuration

Figure 4.1 illustrates the configuration of the LCSR testbed. All workstations in the LCSR are connected to the Ethernet subnet and also to the ATM switch (using a separate subnet). The machines are linked to a file server (a Sun SparcStation2 workstation) located outside the laboratory, which is also connected to the department's "Research" subnet. The file server also acts as a domain name server and router for the LCSR workstations when they communicate with other machines outside the laboratory. There are currently four machines from the department's "Ariel" subnet sharing the ATM switch (but not the ATM subnet). We ensure that they do not generate any traffic while experiments are being conducted.

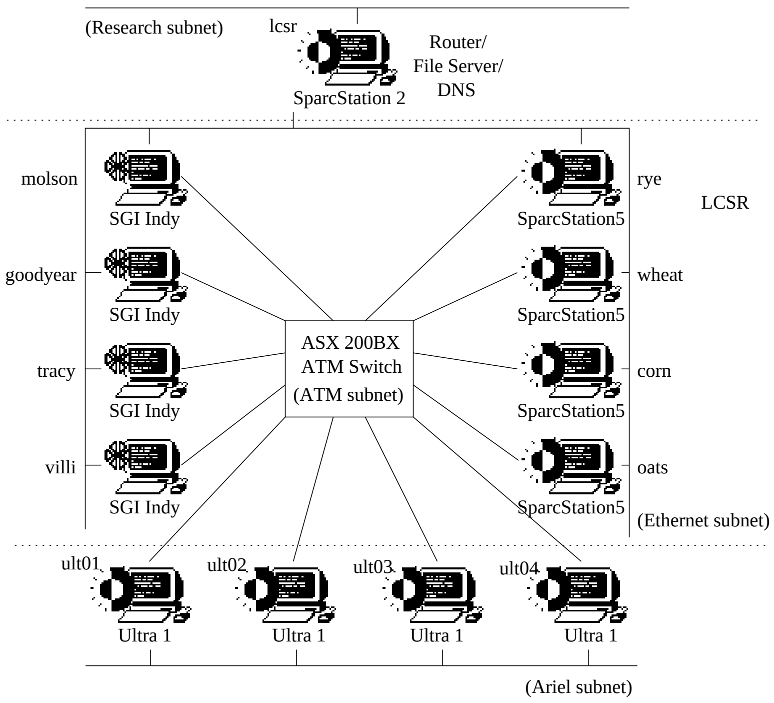


Figure 4.1: The LCSR testbed

4.2 Outline of the Experiments

Before studying the network applications, we need to understand the behaviour of the underlying network environments and the file systems. Our experiments are therefore divided into three phases.

We first conduct memory-to-memory transfer experiments designed to measure the peak throughput of each of the network environments being considered. We identify the sets of software configurable parameters which achieve the best performance (we call these combinations of parameters the "optimal parameters"). In the second phase, we measure the file access rates in order to gain a better understanding of the performance of the file systems. The third phase involves evaluating the performance of the HTTP and FTP applications using each of the three network environments.

Test environment

All experiments are conducted on an unloaded network (i.e., communication is minimized among the machines other than those involved in the experiments). Ideally, the measurements would have been conducted in isolation on dedicated systems, but unfortunately, this is not possible for our machines. The two Sun's named *oats* and *wheat* and the two SGI's named *goodyear* and *tracy* are used. All experiments are carried out on homogeneous platforms (i.e., between the SGI's and between the Sun's, which are the only two platforms available to us).

Timing

Several benchmark programs are used throughout the experiments, and all are capable of measuring the throughput of data transfer based on the transfer time (i.e., the time required to transfer the data). However, the period of time is measured in different ways by the different programs. It is therefore necessary to clarify these differences.

Figure 4.2 illustrates the different ways the timers of these benchmarks measure the transfer time. The "socket/ATM open" refers to the time just before invoking the `socket()` call (for a socket) or the `atm_open()` call (for an ATM_endpoint). The "connection established" refers to the time immediately after the `accept()` or `atm_accept()` call returns on the server, or the `connect()` or `atm_connect()` call returns on the client. The "close connection" is the time immediately after the function responsible for terminating the connection (`close()`, `shutdown()`, or `atm_close()`) returns on the server or the client. Note that the TTCP benchmark uses two timers, one at each end of the connection, but the other benchmarks use only one timer. The rest of this section explains the figure in more detail.

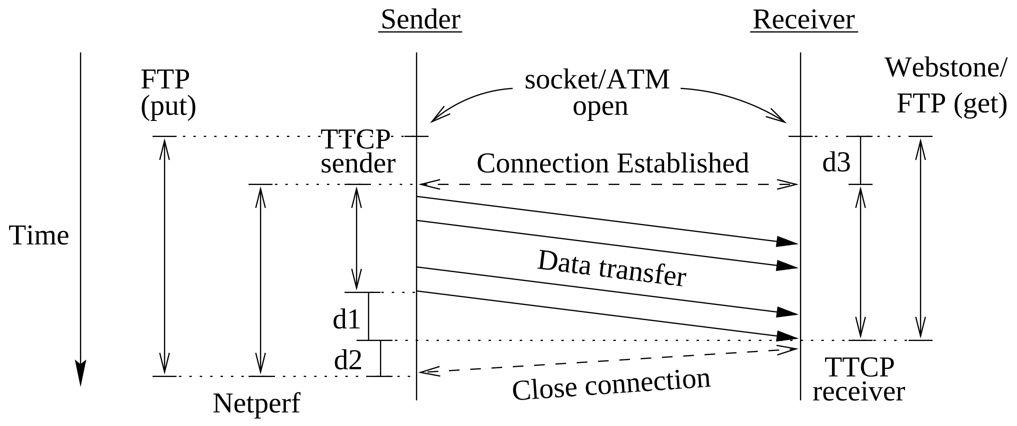


Figure 4.2: Measuring data transfer time using different benchmarks

As mentioned, the TTCP benchmark [64] uses two timers, one at each end of the connection. Each timer records the transfer time from its own side, from the establishment of the connection to the completion of the data transfer. The sender does not wait for the receiver to receive all the data before stopping the timer. Hence, the time reported by the sender is not an accurate measurement of the throughput. This time difference between the sender and the receiver is represented by d1 in Figure 4.2.

The Netperf benchmark [21] measures the transfer time on only the sender side (the client side of the connection). The transfer time is measured from the time the connection is established until it is closed. Netperf ensures that the sender's timer stops only after the receiver has received all the data and has closed the connection, and this is done by invoking the `shutdown()` call on the sender side. However, there is no function call equivalent to `shutdown()` in the Fore API. Therefore, in the native ATM implementation, the sender does not wait for the data to reach the receiver. Thus, the transfer time it reports is not an accurate measure of the throughput (note that the native ATM environment is unreliable, and any throughput reported may be inaccurate).

The Webstone benchmark [62] measures the transfer time of the entire HTTP connection on the client side, from immediately before invoking the `socket()` call until immediately after closing the connection (note that this duration also includes the HTTP request, which is not shown in the figure).

Our modified version of the FreeBSD FTP program (see Section 3.7.5) measures the transfer time of the entire FTP data connection on the client side, excluding the control connection. In order to accurately measure the time of the FTP "put" transfer session, it is necessary to ensure that the timer of the FTP client (the sender) does not stop before the other end has received all data and closed the connection. Therefore, we apply the same technique used by Netperf, which involves the `shutdown()` call.

In a preliminary test, we investigate the significance of the time durations shown in Figure 4.2, which include:

- d1 — the difference between the sender and the receiver's transfer times,
- d2 — the time difference between both sides, from establishing the connection to closing it, and
- d3 — the time duration from the `socket()` (or `atm_open()`) call to the connection establishment.

We generally found that the time durations of d2 and d3 are not significant, but that d1 may take a significant amount of time. Thus, we conclude that the measurement taken from receiver side, from the time the connection is established until closing the connection, is accurate to represent the transfer time. The same applies to the sender's side only if its timer stops immediately after all the data is received by the other side.

In our experiments, we obtain the throughput by dividing number of bytes transferred by the overall transfer time, which includes the time it takes for the data to travel through the kernel and the hardware. Hence, throughput is obtained by measuring the "latency" of the entire transfer. As a result, the time required to perform each transfer can be derived by multiple the throughput obtained by the number of bytes transferred.

To be consistent, all throughput measurements in this thesis are reported in megabits per second (Mbps), which is equivalent to 1024×1024 bits per second. Since the "throughput" metric used in Netperf refers to "Mbps" as 10^6 bits per second, we have rescaled the Netperf measurements to the unit we have used.

Test parameters

Due to the intractability of testing all combinations of parameters, throughout our experiments we only vary the following network parameters to determine their impact on throughput:

- **Socket buffer size** — this is the size of the send and receive socket buffers. The maximum size permitted by both the SGI's and Sun's is 256 KB. Since the native ATM environment does not implement sockets, this only applies to the Ethernet and TCP/ATM environments.

Under normal circumstances, a call to the function `getsockopt()` can be made to obtain the socket buffer size. The SGI's (running IRIX 6.2) give a default value of 60 KB. Although the Sun's (running Solaris 2.5.1) return a default value of "0", we believe its STREAMS implementation uses the `tcp_xmit_hiwat` parameter for the same purpose (this is confirmed in our memory-to-memory transfer experiments), and its default is 8 KB (when it is not specified by the user). In the rest of the chapter, we refer to the "default socket buffer size" on the Sun's as the one which results when no socket buffer size is specified; we believe this default size is 8 KB. The sizes of the send and receive buffers can be changed using the `setsockopt()` call within the network application. The changing of these sizes is local to the application and therefore does not affect other applications.

- **Data buffer size** — this is the size of data buffer used by the application's own send and receive functions (e.g., `write()`, `read()`, `atm_send()` and `atm_recv()`). The data buffer size is specified within the network application.

- **ATM MTU** — this is the maximum size of the CPCS-PDU (see Section 2.1.4), and applies to the TCP/ATM and native ATM environments only. In the TCP/ATM environment, the data is fragmented if the data size (or data buffer size) is larger than this limit. In the native ATM environment, the use of data (or a data buffer) longer than this limit will result in an error. Fore Systems provides a default ATM MTU of 9188 bytes and a maximum allowed ATM MTU of 65535 bytes. Since the Fore API implementation requires an extra 4 bytes for header information (see Section 2.3.4), the maximum possible data buffer size allowed in native ATM is actually 65531 bytes. TCP/ATM does not impose a limit (besides the limit imposed by the network input/output calls) because additional fragmentation can be done in the TCP and IP layers if needed. The ATM MTU size affects the entire system and can only be altered with root privilege to the system.

- **ATM window size** — because the design of ATM networks and protocols does not ensure reliable data transfers between hosts, we implement a simple reliable protocol for use with the native ATM applications (see Section 3.1.3). The ATM window size refers to the amount of data the sender transmits before waiting for the receiver's acknowledgment.

The 1500 byte Maximum Transmission Unit (MTU) specified by the Ethernet Standard [20] has long been widely accepted and used. In addition, results from the memory-to-memory transfer experiments (see Section 4.3.2) show that the peak throughput achieved using the Ethernet is already near to the theoretical upper bound (84.4% and 91.7% of the theoretical upper bound for the SGI's and Sun's respectively, not including the operating system overhead). Therefore, we decided not to vary the Ethernet MTU size in our study.

Under the default TCP options, acknowledgments from the receiver are delayed until they can be piggy-backed onto either a data segment or a window update packet [15]. This TCP delay option can be disabled by setting the `TCP_NODELAY` flag. However, we believe that by doing this (i.e., with the absence of the delay acknowledgment scheme), more acknowledgments are returned from the receiver, and this cannot possibly improve the throughput. Our memory-to-memory experiments found that the absence of TCP delay option (when the `TCP_NODELAY` flag is used) did not improve the peak throughput in which we are interested. Hence, its effect on throughput is not investigated further.

4.3 Memory-to-Memory Transfer

The first series of experiments measures the rates at which data can be transfered from the memory of an application on one host to that of another. We call these "memory-to-memory transfer" experiments.

4.3.1 Methodology

We conduct the following two sets of experiments on each platform using each of the three environments:

- **Socket buffer test** — we first select a few data buffer sizes and run tests on a full range of socket buffer sizes, from 1 KB to 256 KB, with a 1 KB increment. The same size is used for the send and receiver socket buffers, since this is shown to achieve the best TCP/IP throughput [40]. We select a 1 KB data buffer to represent a comparatively small data buffer size, and the maximum 256 KB data buffer for a comparatively large one. For intermediate sizes, 8 KB is selected, since this size is commonly used in many applications, including the NCSA HTTPd [36] and Webstone [62] applications used in later experiments. Socket buffer tests are not conducted for the native ATM environment since it does not implement sockets.
- **Data buffer test** — different data buffer sizes are then tested using the socket buffer sizes which the preceding test found to be interesting. The default socket buffer sizes are also tested. Data buffer sizes over 256 KB are presumably not interesting to us, because the the maximum socket buffer size of 256 KB would result in the segmentation of the data. Since we found that using data buffer sizes greater than 64 KB resulted in similar behavior in most cases, we only conduct detailed measurements from 1 KB up to 64 KB (with a 1 KB increment step), unless we observe that the measurements beyond this range exhibit different behaviour.

From the results in each environment on each platform, we select the combination of socket and data buffers (also ATM MTU size for the TCP/ATM and native ATM environments) which yield the best throughput, and use it in the HTTP and FTP transfer experiments. We call this combination the "optimal parameters", but they may not necessarily yield the optimal throughput with the network applications.

Netperf [21] is used as the benchmark, and a duration of 10 seconds (Netperf's default) is used for each run. The measurement data is captured over 20 runs, and all data points shown are an average of these measurements. We refer to "throughput" in the results as the average throughput of these runs. The 95% confidence intervals are also shown in the resulting graphs for every second measurement (in order to make the graphs easier to read). All the memory-to-memory transfer experiments and resulting figures are listed in Table 4.1.

4.3.2 Ethernet

The SGI platform

The results of the socket buffer test using the SGI workstations and the Ethernet are shown in Figure 4.3. An exceptionally low throughput of 0.04 Mbps is obtained when using the 1 KB data buffer and 4 KB socket buffer. We found that switching off the TCP delay acknowledgment (by setting the `TCP_NODELAY` option) can eliminate this anomaly, yielding a throughput of 6.3 Mbps.

Network environment	Platform	Test	Figure
Ethernet	SGI	Socket buffer	4.3
		Data buffer	4.5
	Sun	Socket buffer	4.6
		Data buffer	4.7
TCP/ATM	SGI	Socket buffer	4.8
		Data buffer	4.10
	Sun	Socket buffer	4.11
		Data buffer	4.12
Native ATM	SGI	Data buffer	4.13
	Sun	Data buffer	4.14
Ethernet	SGI, Sun	Netperf vs TTCP	4.15-(a)
TCP/ATM	SGI, Sun	Netperf vs TTCP	4.15-(b)
Native ATM (unreliable)	SGI, Sun	Netperf vs TTCP	4.15-(c)
Native ATM (reliable)	SGI	ATM window	4.16
	Sun	ATM window	4.17

Table 4.1: The memory-to-memory transfer experiments

Using a socket buffer size greater than exactly 3×MSS (MSS is 1460 bytes for Ethernet) can also eliminate this problem. Therefore, it is very likely caused by the throughput deadlock problem on some BSD-based systems [17, 45, 40] (see Section 2.4.2). Note that this is the only experimental result that we found to be influenced by the `TCP_NODELAY` option. Apart from this anomaly, the three data buffer sizes yield very similar throughputs for socket buffer sizes up to about 100 KB.

The 8 KB socket buffer clearly yields the best throughput for all three data buffer sizes. In order to locate the precise peak, we run additional tests in the vicinity of 8 KB. We use data buffer sizes of multiples of the MSS and 1 byte less than multiples of the MSS, since further investigation shows that a jump or drop occurs at the boundary of every MSS. This is because in the BSD systems, the socket buffer size is adjusted by the kernel to fit the next lowest MSS [44]. The results, depicted in Figure 4.4, show statistically similar throughputs obtained using socket buffer sizes from 5×MSS to 6×MSS. For example, a throughput of 8.1 Mbps is achieved by an 8760 (6×MSS) byte socket buffer. This is 84.4% of the theoretical upper bound of 9.6 Mbps for the Ethernet (from Figure 2.2, 1460 bytes out of the total 1518 bytes within the Ethernet frame are user data, and Ethernet bandwidth is 10 Mbps. Hence, 10×1460/1518 Mbps=9.6 Mbps).

In the corresponding data buffer test (Figure 4.5), we only show data buffer sizes of up to 64 KB (the region beyond this range exhibits similar behaviour). Three socket buffer sizes are considered: 60 KB which is the default under IRIX 6.2, the 256 KB maximum, and the observed optimal 8760 bytes obtained in the previous experiment. The results show no significant change in throughput with different data buffer sizes. This shows that the impact of the data buffer size is overridden by that of the underlying socket buffer size. As expected, the best throughput is obtained with the 8760 byte socket buffer. We do not know the precise reason that this size yields the best throughput.

The Sun platform

Figure 4.6 shows the results of the socket buffer test using the Sun platform. The original vertical scale of the graphs shows rather steady behaviour, so it is enlarged to reveal more detail. We run additional tests using the 4 KB data buffer which yields the peak throughput in the corresponding data buffer test (Figure 4.7). The results of the socket buffer test indicate that the 4 KB data buffer size performs slightly better than the 8 KB data buffer, but the 256 KB data buffer is as

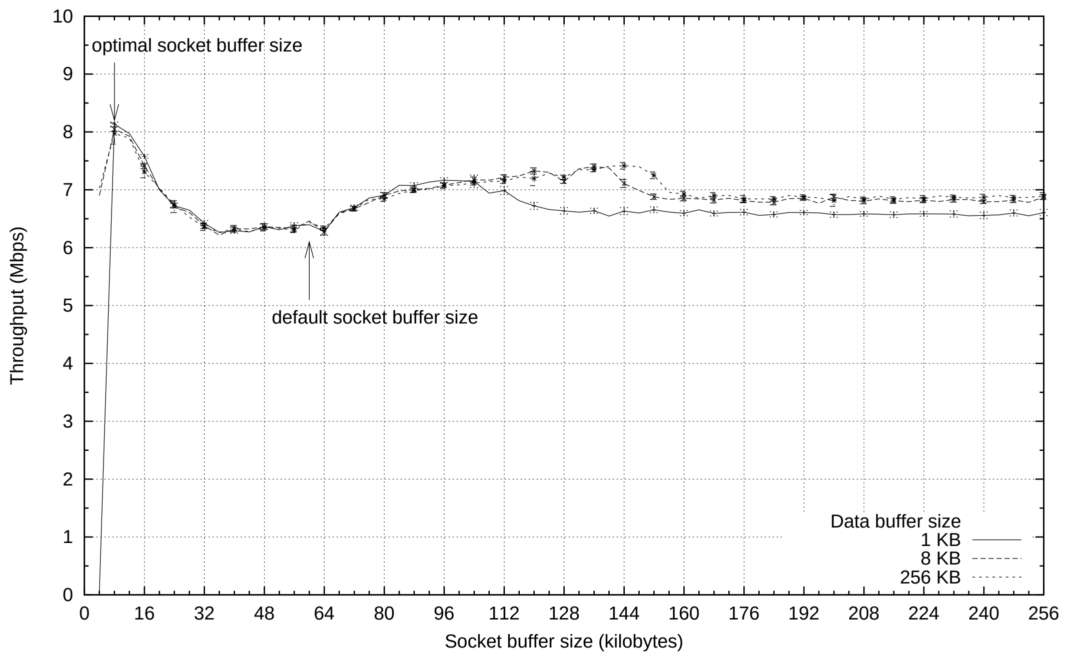


Figure 4.3: Netperf memory-to-memory transfer (Ethernet, SGI) — socket buffer test

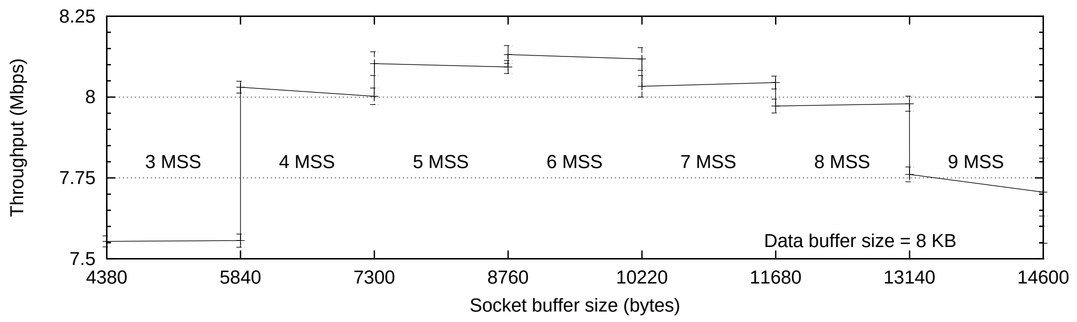


Figure 4.4: Netperf memory-to-memory transfer (Ethernet, SGI) — peak throughput

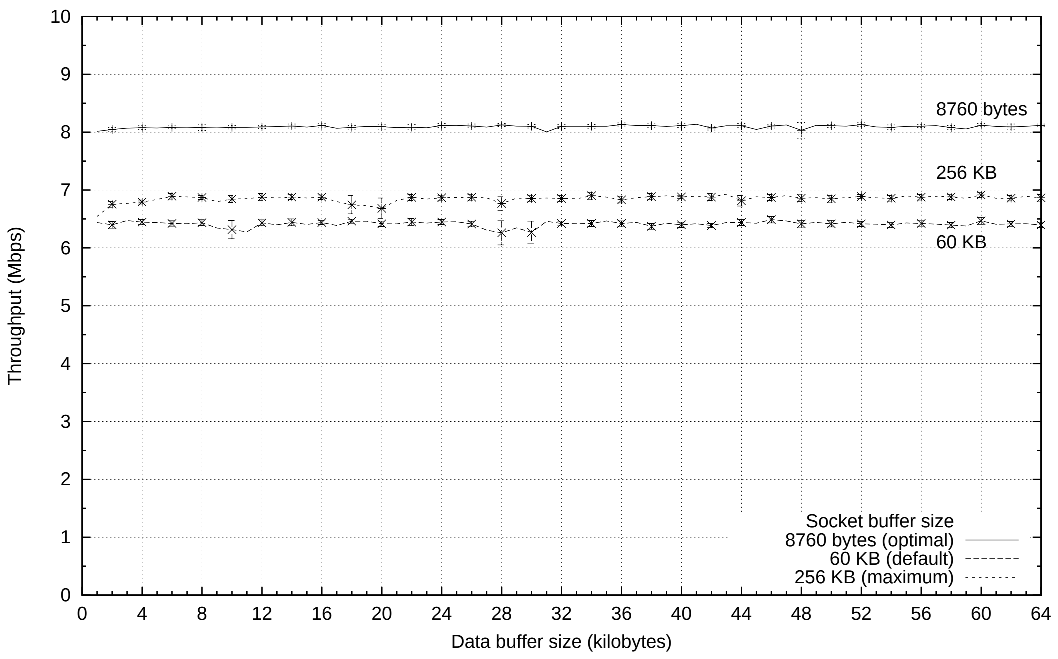


Figure 4.5: Netperf memory-to-memory transfer (Ethernet, SGI) – data buffer test

good, and may even be better in some cases.

Figure 4.7 shows the results of the corresponding data buffer test. Again, the vertical scale of the graph is enlarged. We show the throughputs obtained using the default, the maximum 256 KB, and the intermediate 64 KB socket buffer sizes. The results indicate that the default socket buffer size performs the slowest among the three sizes examined. The 64 KB socket buffer performs marginally better than the 256 KB socket buffer, and both socket buffer sizes yield the best throughputs with a 4 KB data buffer.

A peak throughput of 8.8 Mbps is achieved using a 4 KB data buffer size and 64 KB socket buffer, which translates to 91.7% of the theoretical upper bound of 9.6 Mbps.

The results on both platforms using the Ethernet environment indicate that, with suitable adjustments to the software configurable parameters, the network performance can be improved significantly. For example, the results of the socket buffer test on the SGI's show that throughput is improved by a factor of 1.25, from 6.5 Mbps using the default 60 KB socket buffer to 8.1 Mbps using a 8760 byte socket buffer. It will be interesting to see whether such improvements can also be achieved by the network applications.

The above results also show that when using the Ethernet environment, the throughputs obtained on the Sun's are generally better than those obtained on the SGI's. However, we will see that the SGI's perform better than the Sun's in the other two network environments.

Optimal parameters

On the SGI's, the peak throughput of 8.1 Mbps is obtained when using the 8760 byte socket buffer. Throughput is generally unaffected by the use of various data buffer sizes. Thus, there is no need to use an overly large data buffer (8 KB is a good choice). Indeed, this combination of optimal buffer sizes is shown (see Sections 4.5.2 and 4.6.2) to out-perform the default buffer sizes

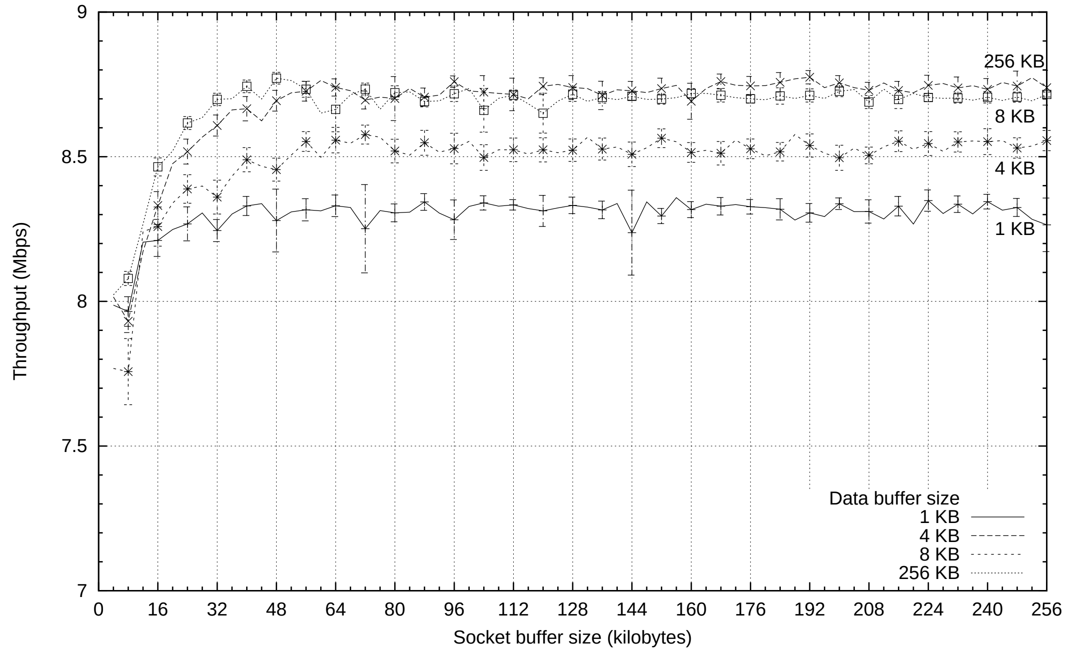


Figure 4.6: Netperf memory-to-memory transfer (Ethernet, Sun) – socket buffer test

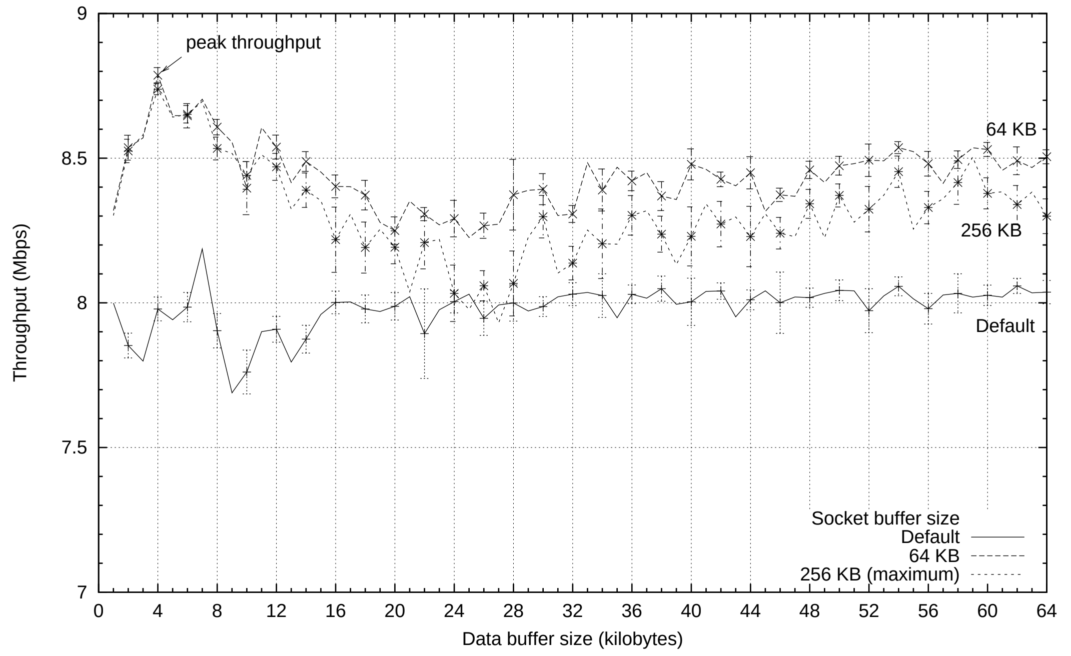


Figure 4.7: Netperf memory-to-memory transfer (Ethernet, Sun) – data buffer test

in the HTTP and FTP experiments using the Ethernet environment.

The use of the 4 KB data buffer and 64 KB socket buffer on the Sun's yields the best throughput of 8.8 Mbps, and thus they are chosen as the optimal parameters. Note that this is only marginally better than the throughput obtained when using the 8 KB data buffer. The combination of these buffer sizes is shown (see Sections 4.5.2 and 4.6.2) to out-perform the default buffer sizes in most of the HTTP and FTP experiments using the Ethernet environment.

4.3.3 TCP/ATM

In the TCP/ATM experiments, we conduct tests using the default (9188 bytes) and maximum (65535 bytes) ATM MTU sizes. Since the TCP MSS is adjusted to fit the underlying MTU [50] (ATM MTU in the case of TCP/ATM), its size is directly influenced by the ATM MTU size. Hence, we expect the two ATM MTU sizes to exhibit different throughput behaviour in our experiments.

The SGI platform

Figure 4.8 shows the results of the socket buffer test on the SGI platform. Using the default ATM MTU, all three data buffers yield a dramatic stepwise increase in throughput as the socket buffer size increases until a maximum is reached (steps occur at 8 KB intervals). Using the maximum ATM MTU, the 256 KB data buffer results in another stepwise curve with steps occurring at exactly 120 KB, 180 KB (a small and not very obvious step), and 240 KB. Further investigation of the relationship between the ATM MTU and the MSS reveals that the BSD-based kernel rounds up the MSS to 8 KB for the default ATM MTU and to 60 KB for the maximum ATM MTU, both corresponding to the highest multiple of 8 KB which is less than the ATM MTU size. The BSD sockets implementation further adjusts the requested socket buffer size to become divisible by this MSS [44], as illustrated in Figure 4.9 (this does not happen in the STREAMS-based Solaris systems). Therefore, instead of gradual increases in throughput, stepwise increases are observed.

The peak throughput occurs with socket buffer sizes between 240 KB and 256 KB, yielding a peak throughput of 121.4 Mbps; this is 87.1% of the theoretical upper bound of 139.4 Mbps (see Section 2.4.2). These results show that the throughput generally increases with the socket buffer size (after being adjusted by the kernel). By comparing the two ATM MTU sizes using the 256 KB socket buffer (Figure 4.8), the peak throughput is improved from 114.3 Mbps using the default ATM MTU to 121.4 Mbps when using the the maximum ATM MTU.

In the corresponding data buffer test (Figure 4.10), we use socket buffer sizes of 60 KB (the default) and 256 KB (the maximum) with the default ATM MTU, as well as with the maximum ATM MTU of 256 KB. We examine data buffer sizes ranging from 1 KB to 256 KB in increments of 4 KB (we found that in this case the 256 KB socket buffer and default ATM MTU exhibit different behaviours beyond the 64 KB data buffer size). We use an increment of 1 KB for data buffer sizes between 1 KB and 64 KB for the maximum ATM MTU curve to reveal greater detail in the region. The default socket buffer size achieves a peak throughput of 99.3 Mbps using a data buffer size of 16 KB. The peak throughput obtained with the 256 KB data buffer using the default and maximum ATM MTU's is 113.2 Mbps and 121.4 Mbps respectively. This confirms the results from the previous experiment that higher throughput is obtained using the maximum ATM MTU.

The stepwise jumps in throughput observed in the results indicate that the choice of the data buffer sizes has a big impact on throughput. For example in Figure 4.8, a 236 KB data buffer yields a throughput of 87.2 Mbps, whereas a 240 KB data buffer yields a throughput of 121.9 Mbps.

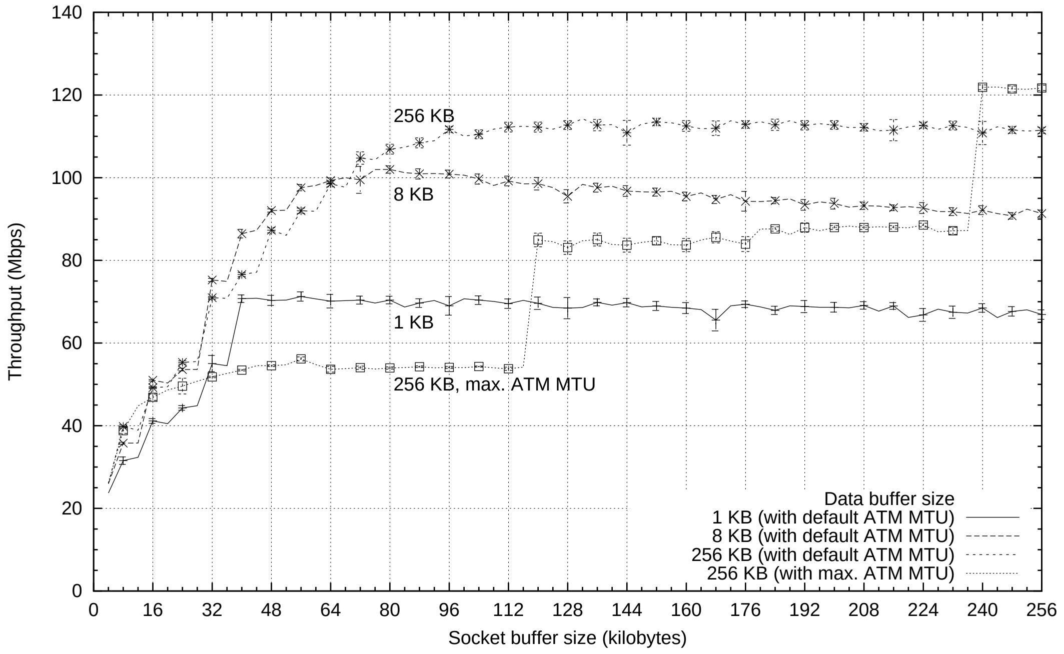


Figure 4.8: Netperf memory-to-memory transfer (TCP/ATM, SGI) — socket buffer test

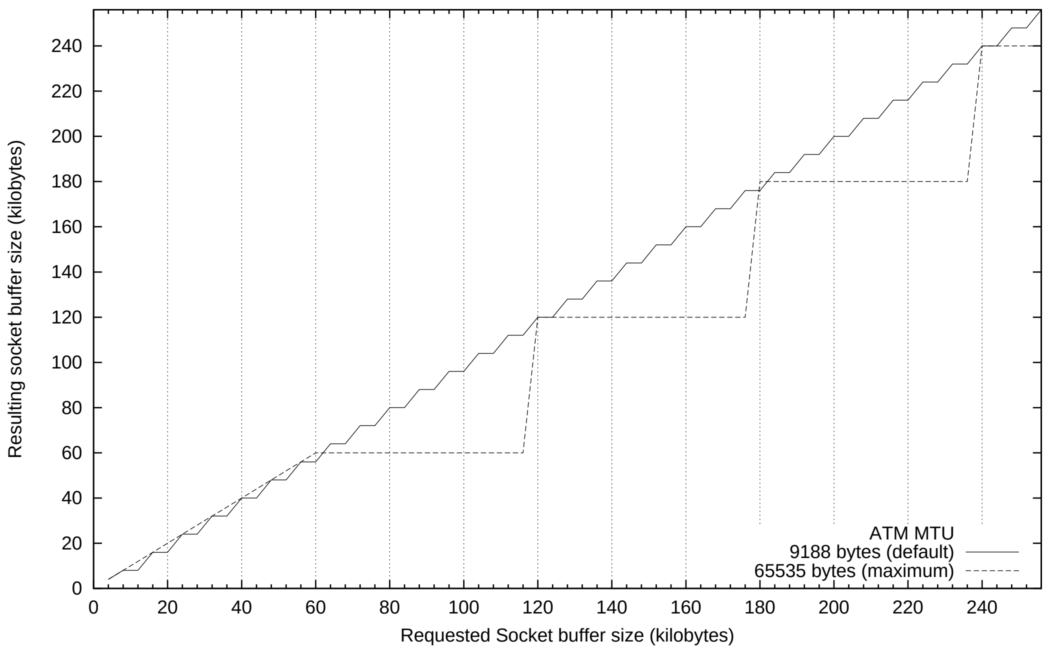


Figure 4.9: Adjustment of socket buffer size under BSD Sockets

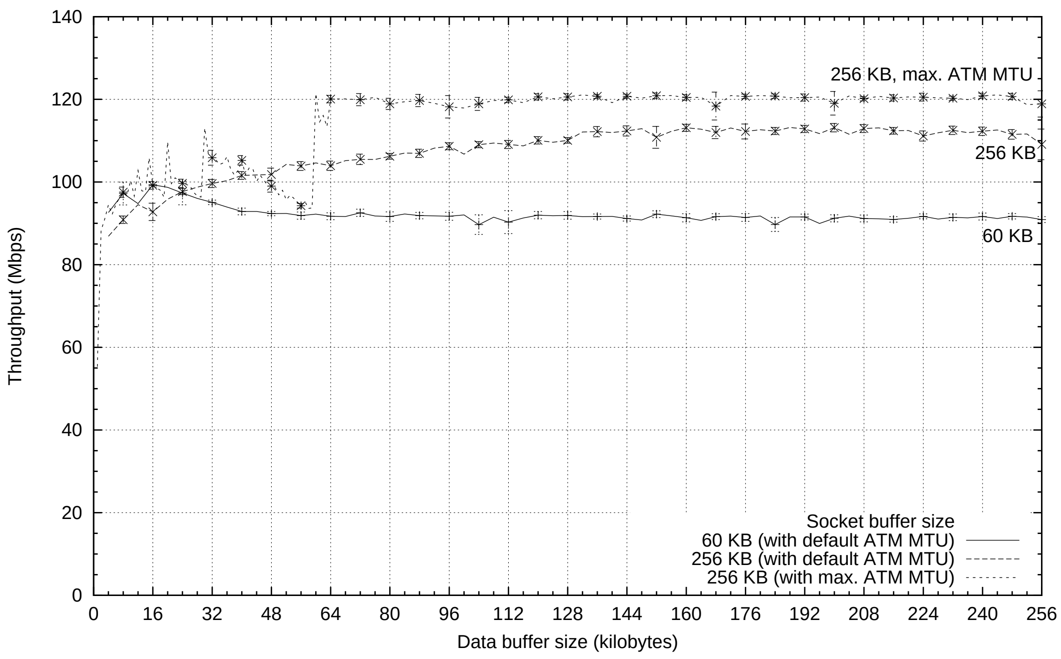


Figure 4.10: Netperf memory-to-memory transfer (TCP/ATM, SGI) — data buffer test

The Sun platform

Figure 4.11 depicts the results of the socket buffer test on the Sun's. Again, larger data buffer sizes generally result in better throughput, which is expected due to the decrease in data fragmentation. From the next experiment, we find that a 16240 byte data buffer yields the best throughput using the maximum ATM MTU (Figure 4.12), but not with the default ATM MTU. Therefore, this data buffer with the maximum ATM MTU is also tested in this experiment. As shown in the results, it achieves throughputs equal to or better than those of the 256 KB data buffer with the default ATM MTU.

Unlike the corresponding results on the SGI's, the socket buffer size does not have a big impact on throughput on the Sun's. In addition, no stepwise increase in throughput is observed, because the Sun's kernel adjusts the MSS to the ATM MTU size, but it does not round it down to a multiple of the MSS, as is done on BSD systems.

We also conduct a socket buffer test on a 64 KB data buffer, which yields throughputs very similar to those of the 256 KB data buffer. Therefore, we do not investigate beyond the 64 KB data buffer size in the corresponding data buffer test, whose results are illustrated in Figure 4.12. The graph reveals some "saw-tooth" structures in all curves. From further investigations, we learn that the peaks occur at multiples of the MSS for the default and maximum ATM MTU's being tested (in the TCP/ATM environment of the STREAMS-based systems, an MSS is equal in size to the ATM MTU minus 40 bytes, see Section 2.1.4). These periodic drops are caused the use of extra CPCS-PDU's, and not by the amount of CPCS-PDU padding used, as observed by Luckenbach *et al.* [40] and Fouquet *et al.* [26] (as described in Section 2.4.3). Using the maximum ATM MTU, additional drops in throughput occur before 16 KB and 32 KB. Further investigation locates the drops at precisely after 16240 and 32747 bytes. We do not know the exact cause of these drops, and it is likely that these drops are caused by the overhead introduced due to the use of extra

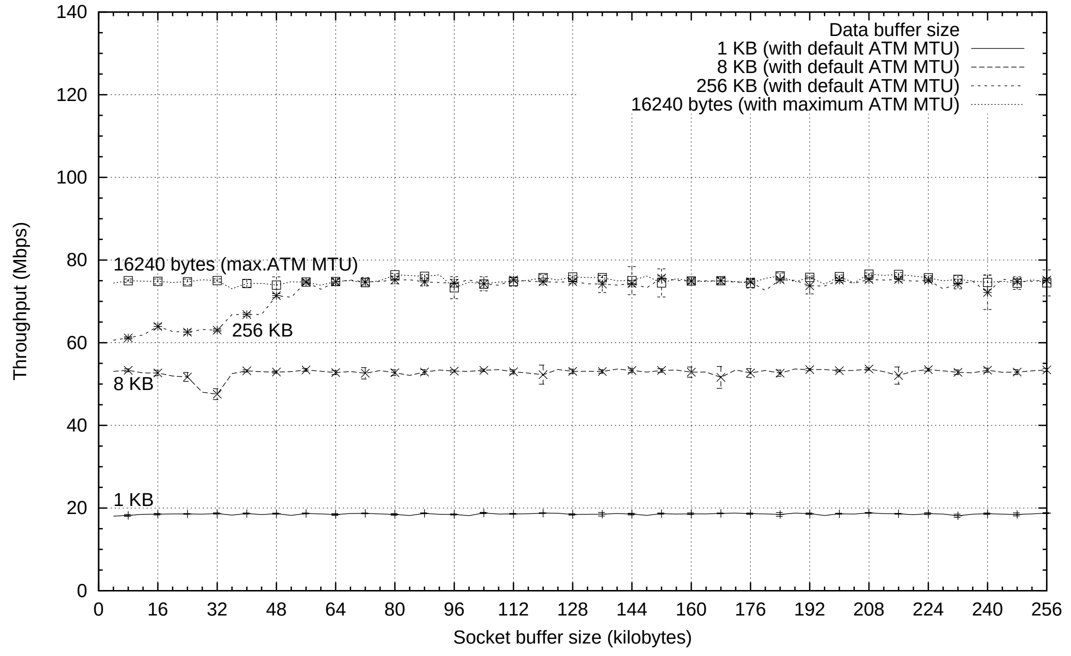


Figure 4.11: Netperf memory-to-memory transfer (TCP/ATM, Sun) — socket buffer test

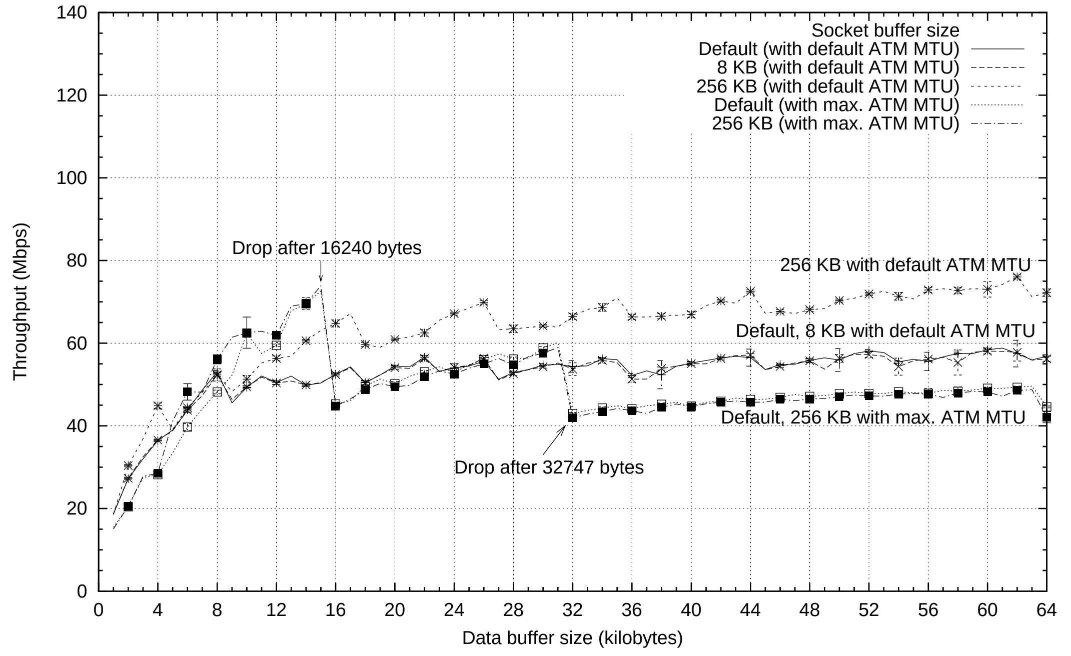


Figure 4.12: Netperf memory-to-memory transfer (TCP/ATM, Sun) — data buffer test

buffers to accommodate additional data (the gradual rises after the drops indicate the reduction of this overhead, as the extra buffer is being filled by additional data from the increasing data buffer). We suspect that these drops are caused by the use of some 16 KB message blocks in the Fore TCP/ATM implementation (the values are slightly less than the multiples of 16 KB, because some bytes are used to store the message block headers and other information).

When using the maximum ATM MTU, both the default and maximum socket buffer sizes yield statistically similar throughput. A peak throughput of 75.7 Mbps (54.3% of the theoretical upper bound) is observed with the 16240 byte data buffer using the maximum ATM MTU.

The presence of this "saw-tooth" behaviour indicates that the choice of the data buffer size is very important for achieving good performance. For example, using the maximum socket buffer size and ATM MTU, the 16240 byte data buffer yields a throughput of 75.7 Mbps, while a slightly larger 16384 byte (or 16 KB) data buffer yields a throughput of only 44.8 Mbps.

The curve shown for the default socket buffer sizes matches the 8 KB socket buffer curve, confirming that the 8 KB `tcp_xmit_hiwat` network parameter corresponds to the default socket buffer size under Solaris. This socket buffer size is comparatively smaller than that used by the SGI's. We also observe that in this environment, significantly lower throughput is generally obtained on the Sun's than on the SGI's.

Optimal parameters

The TCP/ATM experiments show that a peak throughput of about 121 Mbps is achieved on the SGI's using the maximum ATM MTU and the maximum socket buffer size, as well as a data buffer of size 60 KB, 64 KB or larger (Figure 4.10). We choose 60 KB to be the optimal data buffer size for the HTTP and FTP transfer experiments. The maximum ATM MTU is also selected. Indeed, it is shown in the subsequent experiments to perform better than the default ATM MTU.

On the Sun's, two peak throughputs are observed: the 64036 byte data buffer using a 256 KB socket buffer and the default ATM MTU, and a 16240 byte data buffer using the maximum ATM MTU and either the default or 256 KB socket buffer (Figure 4.12). We choose the latter for use in the HTTP and FTP experiments, firstly because it is smaller (consuming less resources), and secondly, because it performs just as well with the default and maximum socket buffer sizes. For this reason, we also use the default socket buffer size.

4.3.4 Native ATM

Only the data buffer tests are conducted using the native ATM network environment, since it does not utilize socket buffers. Data buffer sizes of up to the maximum 65531 bytes are allowed on the Sun's. However on the SGI's, only sizes up to 32763 bytes can be used, due to a limitation in the IRIX STREAMS implementation. A variable, `mi_maxpsz` in `struct module_info`, which represents the maximum packet size of the module is a signed integer (i.e., 2^{15}-1=32767 bytes), with an additional 4 bytes being used for the Fore Systems CPCS-PDU header (as described in Section 2.3.4).

We show only the results that were obtained using the maximum ATM MTU, because we found that the use of the default and maximum ATM MTU's both yield the same results. This is as anticipated because the ATM MTU only imposes a maximum size on the transmission unit, and unlike TCP/ATM, there is no other fragmentation of data packets from a higher layer that depends on the ATM MTU.

Our results show both the sender's and receiver's throughputs reported by the Netperf's Fore API tests. Netperf records the number of bytes received by the receiver, but does not report this information to the user. It does, however, report the receiver's throughput, which is obtained by dividing this number by the sending time (this time is measured as shown in Figure 4.2). In addition, it reports the sender's throughput, which is obtained by dividing the amount of data sent by the same sending time. Both metrics are shown because they indirectly indicate the ratio of the received and sent data. The reason the amount of received data is sometimes less than that of the sent data can be due to data loss during the transmission, memory buffer overflow, or the sender closing the connection before the receiver receives all the data (there is no equivalent `shutdown()` call provided by the Fore API implementation to prevent this from occurring). Here, we consider all these cases as "data loss". Note that in the TCP tests (for Ethernet and TCP/ATM), Netperf only reports the sender's throughput, since TCP ensures reliable delivery of all data [50].

Note that the measurement reported by Netperf called "throughput" in this unreliable network environment is a rather meaningless metric because the data is transferred over an unreliable network environment. However, we report it to ensure the completeness of our experiments, and it is also used as a reference for throughput evaluation in the corresponding reliable network environment (Section 4.3.6).

The SGI platform

Figure 4.13 shows the results obtained using the SGI's and the unreliable native ATM environment. Data loss is observed below 4 KB (in this region, the receiver's throughput is less than the sender's throughput). Above 4 KB, the receiver's throughput matches that of the sender, indicating that no data is lost. Drops occur immediately after 4092 (4 KB−4) bytes, 8188 (4 KB×2−4) bytes, and 12284 (4 KB×3−4) bytes. As described in Section 2.3.4, the ATM adapter card utilizes a finite pool of 4 KB message blocks under IRIX. Accounting also for the 4 byte Fore header, for data buffer sizes slightly larger than 4 bytes less than a multiple of 4 KB, an extra CPCS-PDU is required, resulting in a drop in the throughput. This is not observed for data buffer sizes larger than 16 KB, and it is likely that a different message block size is being used for these sizes. A maximum throughput (for both sender and receiver) of 126.7 Mbps is observed with data buffer sizes between 20 KB and 28 KB.

The Sun platform

The results of the data buffer test using the Sun's (Figure 4.14) show a drop in the receiver's throughput after 15 KB. Detailed investigations indicate that the precise peak is at 16308 bytes, achieving a throughput of 73.4 Mbps.

The difference between the sender's and receiver's throughput indicates that, using a data buffer size less than 16308 bytes, only about half of the data sent is being successfully received (the receiver's throughput is roughly half that of the sender's). We also found that data loss is severe for data buffer sizes of 9 KB, 18 KB, and 27 KB. However, we do not know the exact cause of these losses.

Optimal parameters

The above native ATM experiments show that data loss occurs on the SGI's for small data buffer sizes (less than about 4 KB), and therefore small sizes should be avoided in the network applications. The use of a data buffer size between 20 KB and 28 KB on the SGI's yields the best throughput (24 KB is chosen as the optimal data buffer size).

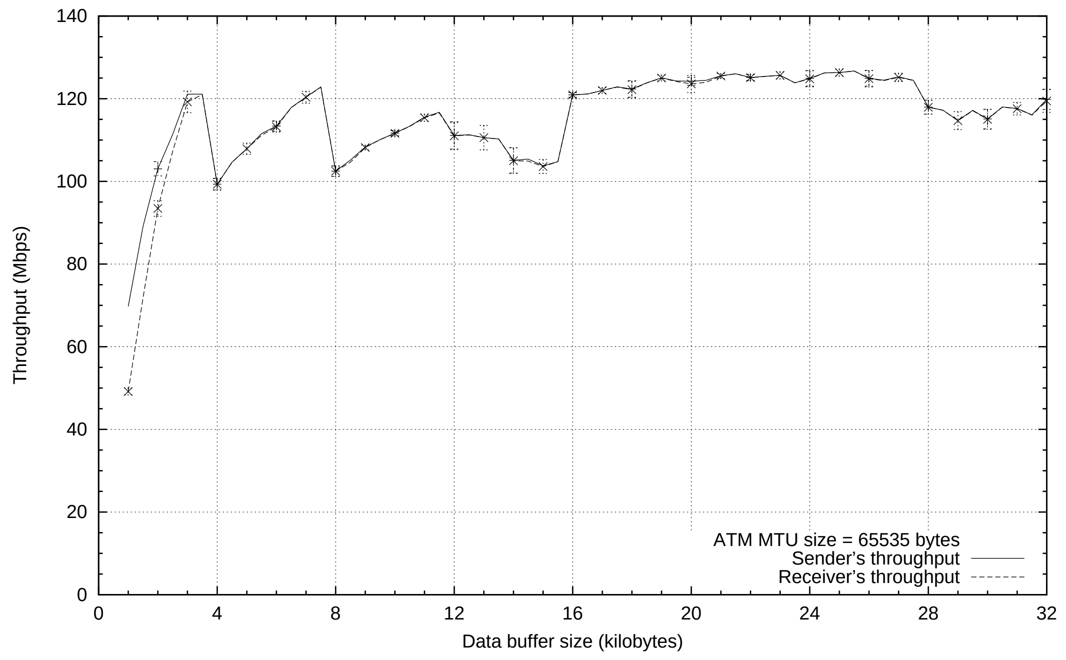


Figure 4.13: Netperf memory-to-memory transfer over unreliable native ATM (SGI)

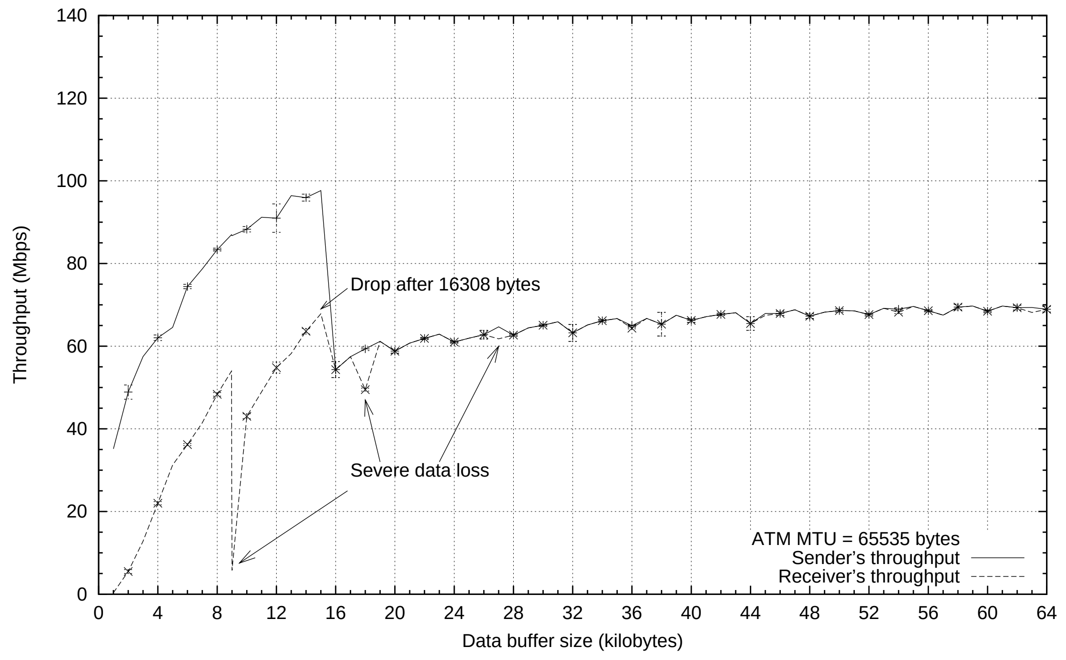


Figure 4.14: Netperf memory-to-memory transfer over unreliable native ATM (Sun)

On the Sun's, some data buffer sizes should also be avoided (e.g., 9 KB, 18 KB, and 27 KB), as the results indicate that they cause severe data loss. The 16308 byte data buffer yields the best receiver throughput of 73.4 Mbps, and is chosen to be the optimal data buffer size.

Note that the optimal data buffer sizes we have chosen are somewhat larger than the typical sizes of 8 KB or 16 KB. It will be interesting to find out if they can significantly improve throughput when used by the network applications.

4.3.5 Comparison of Netperf and TTCP Results

The Fore API test of the current version of Netperf does not support throughput measurements by sending a fixed amount of data, which is required to test our reliable native ATM protocol (as described in Section 3.1.3, our reliable algorithm requires knowing in advance the ATM window size, which is adjusted based on the number of bytes yet to be sent). In addition, Netperf uses a sophisticated transfer mechanism to improve the accuracy of the measurements, and this makes any modification difficult. Another benchmark, TTCP [64], with its simplicity, permits modifications to be made easily. It also allows a fixed amount of data to be transfered in each test. Therefore, we modify TTCP to support our reliable native ATM protocol. The TTCP benchmark is also used in the file access experiments in Section 4.4 because it can perform throughput measurements using input data from files, which Netperf cannot. However, a major disadvantage of TTCP is its incompatibility among different platforms, making it unsuitable as a standard benchmark (as described in Section 3.3.2, a modification was required to resolve an incompatibility with the SGI and Sun platforms).

In our experiments, the TTCP benchmark program is modified to support native ATM and our reliable native ATM protocol as described in Section 3.3.5. It is vital for us to ensure that both Netperf and TTCP produce the same throughput results, since both are used throughout our experiments. This is shown in the experiments described in this section, in which the data buffer tests obtained by both benchmarks are compared.

Our Netperf experiments are controlled by a timer to ensure that each test has sufficient time to perform the data transfer. TTCP does not provide this feature, and we therefore need to choose a data size large enough to allow sufficient time to accurately measure throughput. We found that a data size of 15 MB is adequate for our network environments.

Results gathered from both platforms over the three network environments are shown in Figure 4.15. The vertical scale of the Ethernet's results is enlarged to reveal the details. The results of all three network environments indicate that both benchmarks give results which are statistically similar. We are therefore confident that the throughputs obtained by TTCP are as valid as those obtained by Netperf in our testbed.

4.3.6 Reliable Native ATM

To compensate for the unreliable native ATM protocols used by the Fore API, we have added a simple mechanism to TTCP to ensure reliability (see Section 3.1.3). We call this the "reliable native ATM" environment. The algorithm requires a user-defined value, known as the "ATM window size", which governs the amount of data the sender transmits before waiting for the receiver's acknowledgment. In the next two experiments, we attempt to identify the best ATM window sizes for use in our reliable native ATM protocol on each of our two platforms.

Data buffer sizes of 24 KB and 16308 bytes are used for the SGI's and Sun's respectively. These values were shown to yield the best throughputs in our previous experiments using the unreliable native ATM environment (Section 4.3.4). In each run, 100 MB of data is transferred.

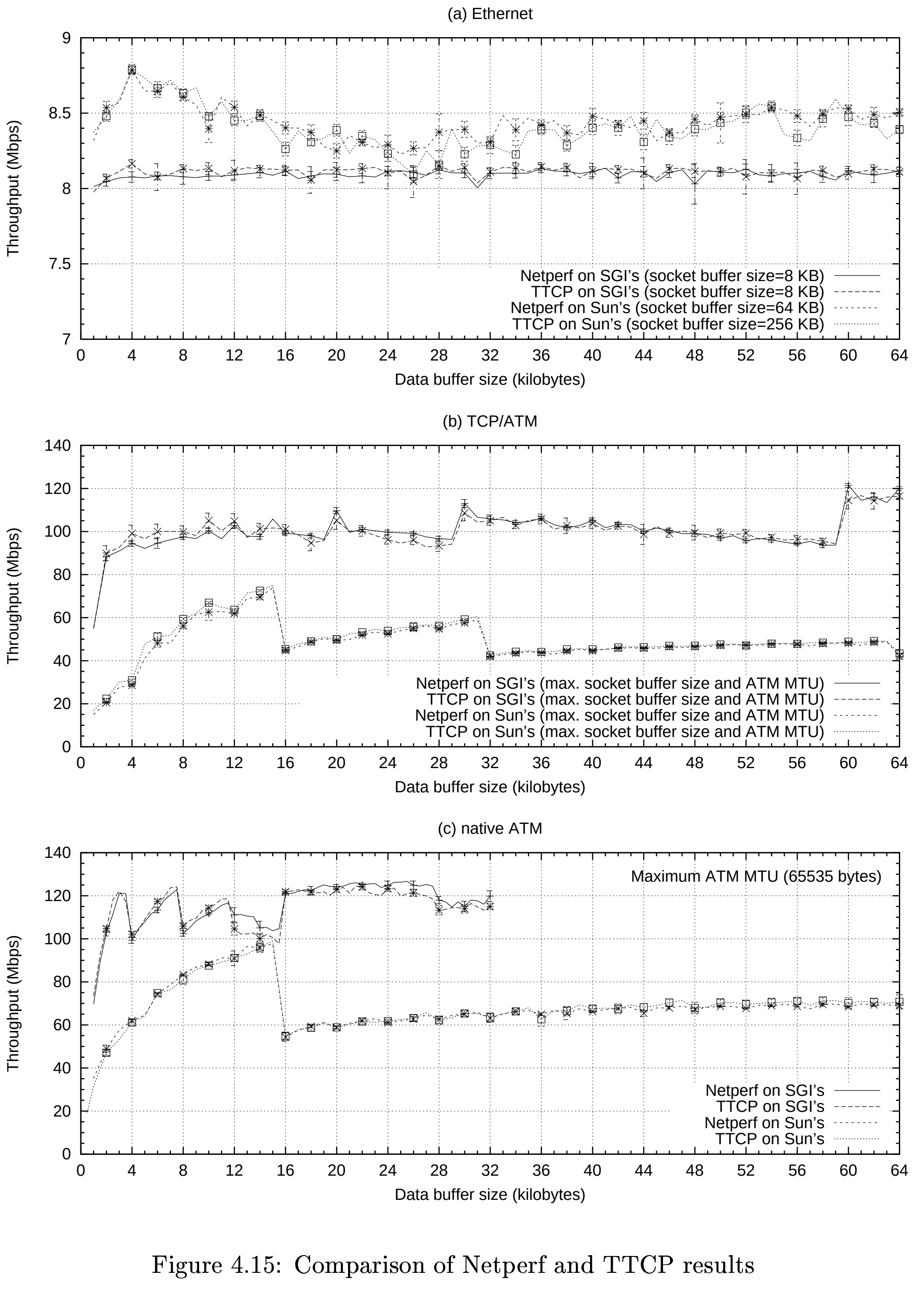


Figure 4.15: Comparison of Netperf and TTCP results

We have deliberately selected a large size to increase the possibility of data loss. The receiver's time-out (the time to wait for additional data before the receiver asks for retransmission) is set to 1 second, which is the minimum time-out value permitted by the `select()` function call used by our reliable protocol. We vary the ATM window size from 100 KB to 2000 KB in 100 KB increments. In addition to showing the average throughput of 20 runs, we also show the throughput of each individual run to reveal the impact of retransmissions. Since the sender's and receiver's throughputs are very similar, we only show the throughputs from the TTCP receiver (which is more accurate, as described in Section 4.2).

The results using the SGI's are depicted in Figure 4.16, and those using the Sun's in Figure 4.17. Both results show slow asymptotic increases in throughput as the ATM window size increases. This is expected, because a larger ATM window size implies fewer sessions and hence fewer acknowledgments to cause delays in the transfer. In our reliable native ATM protocol, data is sent in sessions, with each session ended by a receiver's acknowledgment. See Section 3.1.3 for more details.

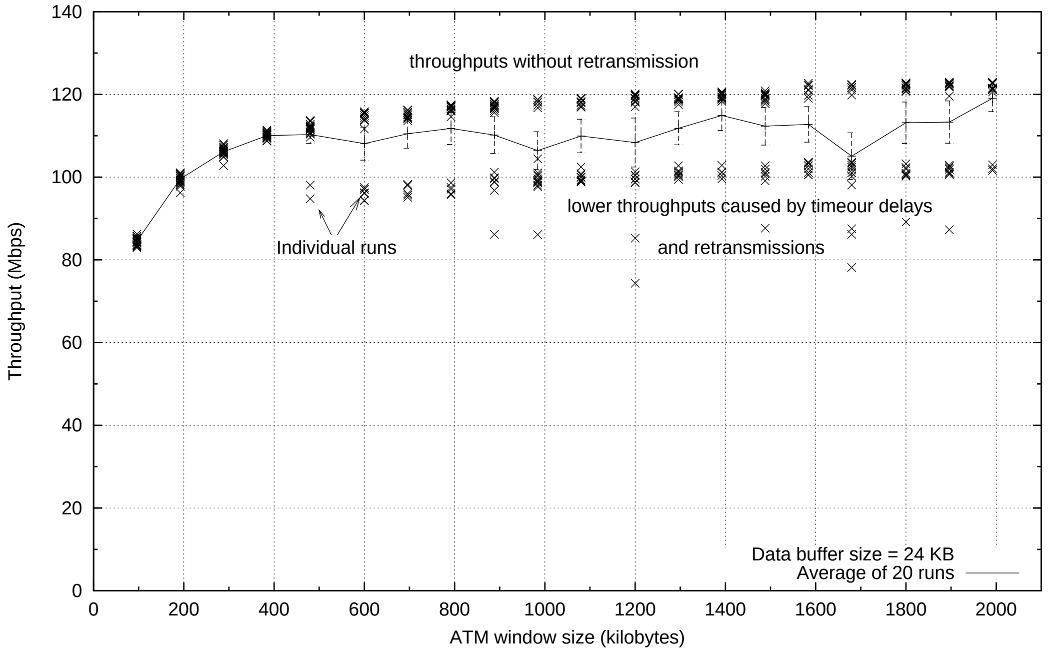


Figure 4.16: TTCP memory-to-memory transfer (SGI, reliable native ATM)

If we observe the throughput obtained in each of the 20 individual runs for each ATM window size, we notice that some have significantly lower throughputs than the others. These low throughputs are caused by the retransmissions (each of these runs contains one or more errors, resulting in extra delays caused by waiting for for the time-out and retransmission). The SGI's generally require more retransmissions as the ATM window size increases. Throughputs without any retransmission are close to the maximum of 121.4 Mbps observed in the corresponding TCP/ATM experiment. Insignificant number of retransmissions is reported on the Sun's using the range of ATM window sizes tested in our experiment.

The maximum average throughput obtained on the Sun's is 81.2 Mbps; this is slightly better than the maximum of 75.6 Mbps observed in the corresponding TCP/ATM experiment. A maximum average throughput of 119.0 is achieved using reliable native ATM on the SGI's, which is statistically similar to the maximum of 121.4 Mbps observed in the corresponding TCP/ATM

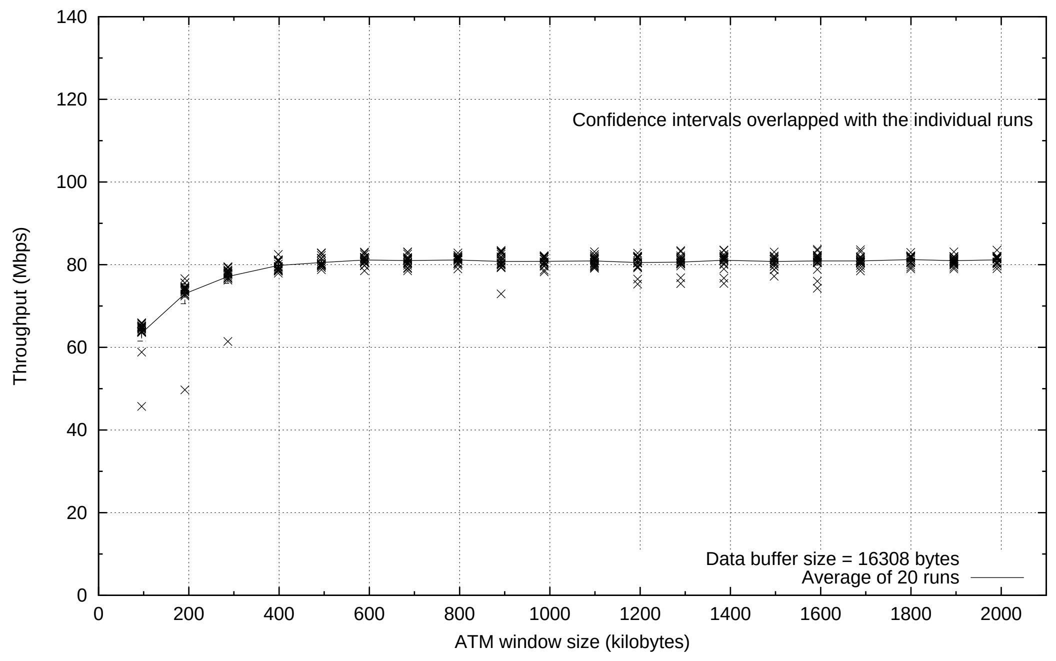


Figure 4.17: TTCP memory-to-memory transfer (Sun, reliable native ATM)

experiment (their confidence intervals overlap each other). These results show that the memory-to-memory transfer in the reliable native ATM environment performs as well as (on the SGI's), or better than (on the Sun's) it does in the TCP/ATM environment.

It should be mentioned that the optimal data buffer sizes being used are based on the throughputs obtained from the unreliable native ATM experiments. These sizes may not yield the best throughputs with the reliable native ATM protocol, but we have decided to use the same size for a fair comparison. The assumption that they also yield the best throughputs with the reliable protocol is justified by the results in which the peak throughputs are very close to those obtained in the unreliable native ATM environment.

Optimal parameters

On the Sun platform, we believe that a peak throughput is reached after the 400 KB ATM window size. It is therefore unnecessary to choose an ATM window size larger than about 400 KB, since we expect that this would only increase the chance of overflowing any memory buffer, and hence increasing the number of retransmissions. In fact, this can be seen in the SGI's results (Figure 4.16), since the number of retransmissions increases as the ATM window size increases.

It is difficult to choose a suitable ATM window size for the SGI's, but as we wish to reduce retransmissions without sacrificing too much performance, 400 KB seems to be an appropriate choice.

4.3.7 Summary of Memory-to-Memory Transfers

The peak throughputs obtained in the memory-to-memory transfer experiments can be regarded as an upper bound for data transfers using each of the network environments. These peak throughput values are referred to in later experiments as an indication of how the performance of a network application is close to its upper limit.

We have done an extensive study on throughput achieved in the three network environments with respect to different socket buffer sizes, data buffer sizes, and ATM MTU sizes. Their characteristics are summarized below:

- **Ethernet** — The Ethernet's results show that throughput on both platforms is greatly limited by the Ethernet network bandwidth of 10 Mbps, where the SGI's and Suns yield peak throughputs which are 84.4% and 91.7% of the theoretical limit (9.6 Mbps), respectively. Thus, any further optimization is not expected to achieve significant improvement.

- **TCP/ATM** — Throughput in the TCP/ATM environment is much higher than in the Ethernet environment. The peak throughputs are 121.4 Mbps on the SGI's and 75.7 Mbps on the Sun's, which are 87.7% and 54.3% of the theoretical limit (139.4 Mbps), respectively.

- **Native ATM** — Using the unreliable native ATM protocol, we observe peak receiver's throughputs of 126.7 Mbps on the SGI's and 73.4 Mbps on the Sun's (recall that, due to the unreliable nature of this test, these throughput values are relatively meaningless). Data loss occurs when using data buffer sizes below 4 KB on the SGI's and 16 KB on the Sun's. Additional data loss occurs with data buffer sizes of 9 KB, 18 KB, and 27 KB on the Sun's. The reliable protocol we have implemented ensures reliable data transfers. The peak throughputs of 119.0 Mbps and 81.2 Mbps are achieved on the SGI's and Sun's, respectively. When these peak throughputs are compared with those of the TCP/ATM environment, it is apparent that the SGI's performance is equal to that of the TCP/ATM environment, while the Sun's achieve slightly better results.

- **Socket buffet size** — The choice of socket buffer size does not have a great impact on throughput when using the Sun's, but it does have a big influence on throughput when using the SGI's. For example, by using the Ethernet and Netperf with 8760 byte send and receiver socket buffers rather than the default 60 KB buffer, the SGI's peak throughput can be improved from 6.5 Mbps to 8.1 Mbps.

- **Data buffer size** — Throughputs obtained using the TCP/ATM and native ATM environments are dramatically influenced by the use of different data buffer sizes, as some resulting throughput curves show "stepwise" or "saw-tooth" shapes. Thus, the choice of data buffer sizes becomes very important for achieving the best possible throughput. For example, using maximum socket buffer and ATM MTU sizes, combined with a 16240 byte data buffer yields a throughput of 75.7 Mbps using TCP/ATM on the Sun's, while the 16384 byte (or 16 KB) data buffer yields a throughput of only 44.8 Mbps.

- **ATM MTU size** — Throughputs obtained using the TCP/ATM environment are greatly affected by varying the ATM MTU size. The maximum ATM MTU size is shown to outperform the default ATM MTU in all our TCP/ATM experiments. The ATM MTU size, however, has no effect on throughput when using the native ATM environment.

We have learned from the experiments that significant improvements in throughput can be obtained by suitably adjusting the socket and data buffer sizes, as well as the ATM MTU size. These improvements are summarized in Table 4.2. The "improvement factor" refers to the ratio of the best throughput obtained by the optimal buffer sizes against that of the default buffer sizes. By "default buffer sizes", we mean the combination of the default socket buffer size and any data buffer size that achieves the best throughput, whereas the "optimal buffer sizes" are the combination of any socket and data buffer sizes that yields the best throughput. Note that there is no default socket buffer size for native ATM, hence no default buffer sizes (the maximum ATM MTU was used but it performed only as well as the default one). With the exception of the TCP/ATM environment on the Sun's, improvement factors of the remaining three configurations are above 1.2, which is quite significant considering that only minor modifications to the software are performed.

	Network environment	Peak throughput (Mbps)		Improvement factor
		Default buffer sizes	Optimal buffer sizes	
SGI	Ethernet	6.5	8.1	1.25
	TCP/ATM	99.3	121.4	1.22
	Native ATM (reliable)	N/A	119.0	N/A
Sun	Ethernet	8.2	8.8	1.07
	TCP/ATM	58.8	75.7	1.29
	Native ATM (reliable)	N/A	81.2	N/A

Table 4.2: Memory-to-memory transfer improvements using the optimal buffer sizes

We have identified the sets of optimal socket buffer, data buffer, and ATM MTU sizes that yield the best throughput for each network environment on each platform. We will use the same combinations of the optimal buffer sizes in later experiments on the network applications.

4.4 File Access

The HTTP and FTP experiments we conduct in later sections require files to be retrieved from and stored to file systems. Therefore, we need to gain a better understanding of each file system's performance in our testbed.

In the following experiments, we measure the file access rates of both the SGI and Sun platforms. We refer to "file read access" as the operation in which a file is read from a local file system, and "file write access" as the operation in which a file is stored to a local file system. This study helps us to understand the limitations imposed by the file systems, and is useful in analyzing the HTTP and FTP experiments in which file transfers can form a central component.

4.4.1 Methodology

Two sets of experiments are conducted: the file read access experiments measure the rate at which the files are read from the disk into an application's memory, whereas the file write access experiments measure the rate at which the files are written from an application's memory to the disk. The effect of file caching is also studied. All experiments in this section use the TTCP benchmark program [64]. To perform file access experiments, the network input/output operations are by-passed completely.

During the experiments, files of specified sizes are read or stored, and the time required to read or write the file is measured. We use the same file sizes which are also used later in the HTTP and FTP experiments (namely 500 bytes, 5 KB, 50 KB, 500 KB, 5 MB, and 50 MB).

The rate at which a particular file is read depends on how it is physically stored on the disk. Disk fragmentation is expected to reduce the access rate, as the disk head is required to read from several physical locations on the disk. In order to compensate for this variation, we use a number of files of each file size, with the exception being that only one 50 MB file is used, due to disk space limitations. All files for the read access experiments are stored in the machine's local directory (`/var/tmp`) to avoid using the network file system. Exactly the same files are used for the later HTTP and FTP experiments (only one file of each size is used in the HTTP experiments).

On both the Sun and SGI platforms, a file cache is maintained by the operating system and is transparent to users. Unfortunately, it can not be easily disabled. For read access experiments using uncached files, we attempt to flush meaningful data from the file cache by accessing another file with a size larger than the file cache size. This should overwrite any contents previously stored in the cache. Since the size of the file cache varies with available system resources, we must ensure that a large enough file is used. Empirically, we have found that a 50 MB file is adequate for both platforms. Hence, we retrieve a 50 MB file (different from the one used for the measurements) in order to "flush" the file cache; this takes place before any of the files is accessed again. This technique, however, does not work well for flushing small files (the 500 bytes, 5 KB, and 50 KB files). Therefore, we ensure that enough files for each of these sizes are available so that each file is accessed only once in each experiment. For experiments using cached files, the same file is read several times, but the first measurement is ignored in the results.

The creation and storage of files sometimes take place within the file cache and may not be immediately performed on the disk. In the write access experiments, an additional `fsync()` function call is invoked after the file is written to the disk, forcing the data in the cache to be written to the disk. We record two stopping times, one before and one after this function, in order to measure the cached and uncached time, respectively. All files are written to the machine's local directory (`/var/tmp`).

Note that the timer called before the `fsync()` function does not necessarily wait for all of the data to be written to the file cache (i.e., the `write()` system call may return after the user data has been transferred to the kernel but before the data has made its way into the file cache). Thus, the access rate measured may be slightly higher than the real value. On the other hand, the measurements obtained for uncached files are accurate, since the `fsync()` call returns only after all of the data has been written to the disk. The results of file write access on cached files are shown to ensure the completeness of our experiments, and to determine how file reads and writes impact the throughput of our applications (e.g., the FTP "get" operation requires reading the file from the server's file system and writing it to the client's file system).

As file accesses do not require socket buffers, we only conduct the data buffer tests. We study a range from 1 KB to 64 KB, as the default and optimal data buffer sizes used in later experiments all lie within this range. We have also chosen an increment of 8 KB (results from additional tests show that no significant difference results from using smaller increments). Table 4.3 lists all the file access experiments conducted, along with their resulting figures.

Platform	Test	Figure
SGI	Read (uncached)	4.18-(a), (b)
	Read (cached)	4.19-(a), (b)
Sun	Read (uncached)	4.20-(a), (b)
	Read (cached)	4.21-(a), (b)
SGI	Write (uncached)	4.22-(a), (b)
	Write (cached)	4.23-(c), (d)
Sun	Write (uncached)	4.24-(a), (b)
	Write (cached)	4.25-(a), (b)

Table 4.3: The file access experiments

The measured data is captured over 20 runs, and each graph shows the average of these measurements along with 95% confidence intervals. For each experiment, we show the same set of results from two perspectives: one for various data buffer sizes, and another for file sizes (with the horizontal axis drawn using a log scale). Note that different vertical scales are used in some graphs.

4.4.2 File Read Access

The SGI platform

Figures 4.18-(a) and (b) show the results of the file read access experiments using uncached files on the SGI platform. In this experiment, the different data buffer sizes do not greatly affect the read access rate. Smaller files have lower disk access rates, as the proportion of file system and disk overheads (e.g., seeks and rotational latency) is greater for these files. This pattern is also observed in the rest of our file access experiments. A peak access rate of about 27 Mbps is obtained using either the 5 MB or the 50 MB files. This suggests that further increases in the file size are unlikely to improve the access rate.

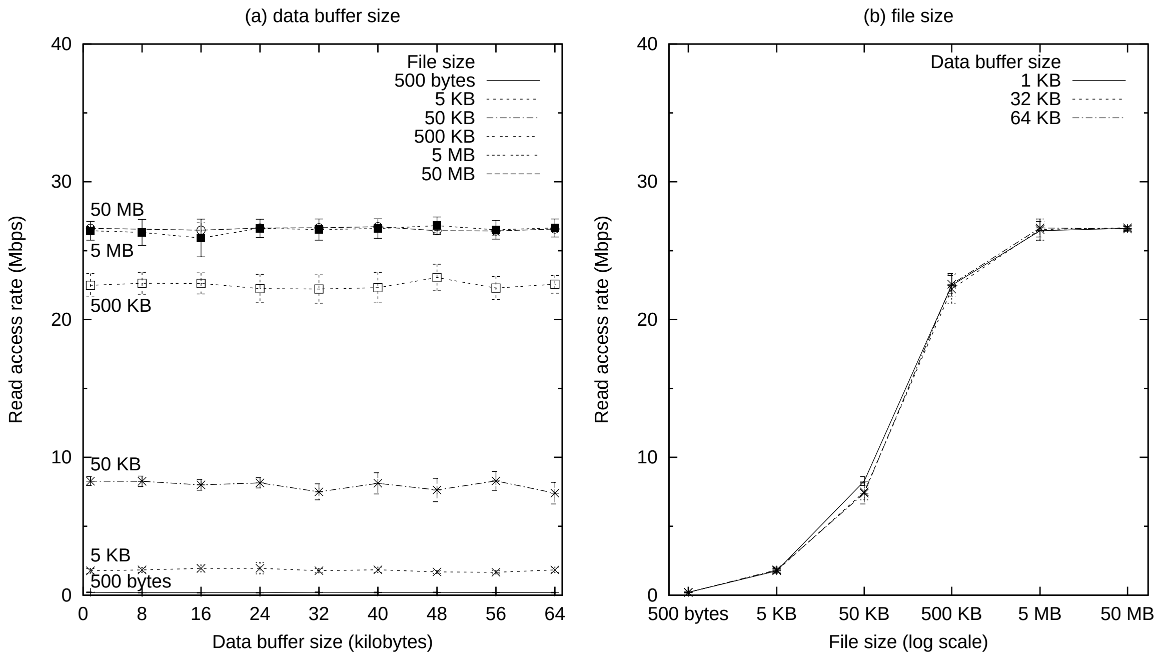


Figure 4.18: File read access rates (SGI, uncached files)

Figures 4.19-(a) and (b) present the results using cached files; these graphs show much higher access rate in comparison with the uncached files, because memory access (the file cache) is much faster than file access. A rather low access rate results when a 1 KB data buffer is used. We conduct further experiments within the range from 1 KB to 8 KB data buffer sizes, and observe a gradual increase in access rate. This suggests that the low access rate is likely caused by data fragmentation resulting from the use of a small data buffer size (as the data buffer size increases, less fragmentation is required, and this reduces the amount of overhead).

While a high access rate is achieved with the 500 KB and 5 MB files, the access rate of the 50 MB file is the same as that obtained in the experiments using uncached files. This indicates

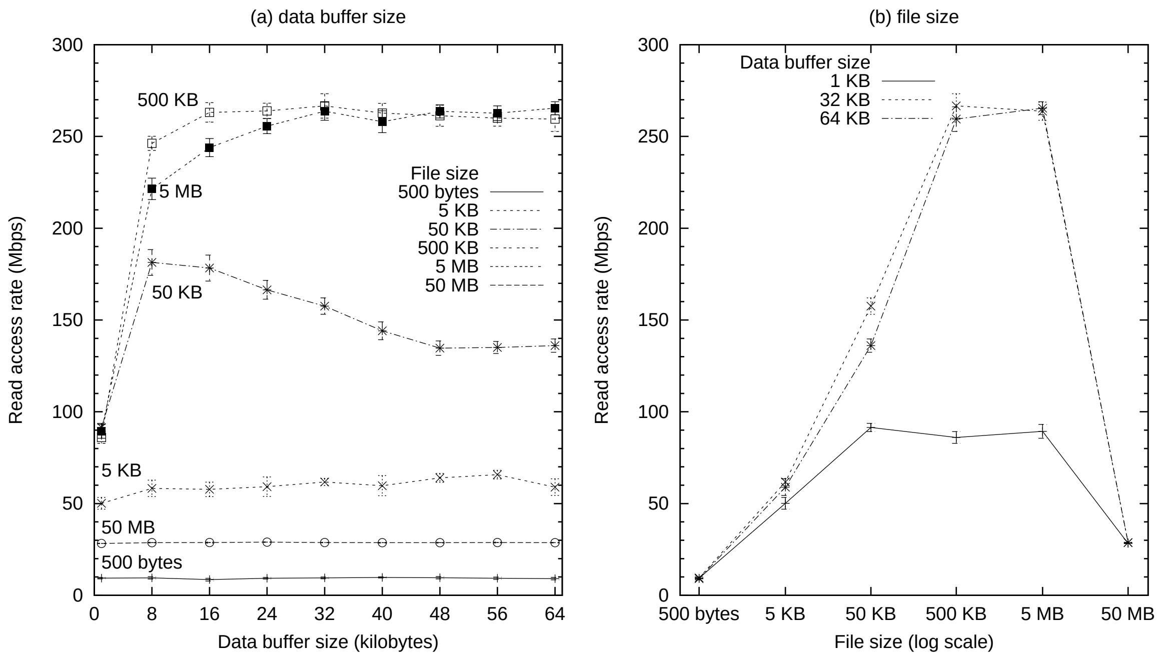


Figure 4.19: File read access rates (SGI, cached files)

that, due to its enormous size which exceeds the cache size, the file is not cached. A maximum read access rate of 266.6 Mbps is obtained using the 500 KB file in the experiment.

The Sun platform

Figures 4.20-(a) and (b) show the results obtained when performing the uncached file read access experiment on the Sun platform. As was the case on the SGI's, the access rate generally decreases with file size. We observe the low throughput obtained when using the 1 KB data buffer, as a data buffer size which is too small causes more data fragmentation, resulting in more overhead. The access rate obtained using the 5 MB file size is slightly higher than that of the 50 MB file size. As the large 50 MB file requires more disk space, it is likely to be more scattered across the disk than the smaller file, and the additional disk head movement causes extra time delays. A maximum access rate of 24.5 Mbps is observed among all the file sizes.

Figures 4.21-(a) and (b) present the results obtained when conducting the same experiment using cached files. Significant improvements are observed, due to the higher memory access speed. Once again, the use of the small 1 KB data buffer size yields low rates for the large files. There is no improvement for the 50 MB file when compared with the results obtained in the uncached experiment, indicating that the cache is not large enough to store the file. A maximum access rate of 164.0 Mbps is observed among all the file sizes.

On both platforms, throughput obtained for uncached files with small file sizes (500 bytes, 5 KB, and 50KB) is comparatively lower than that obtained with larger file sizes (5 MB and 50MB). In addition, the use of a small data buffer (less than 8 KB) is not desirable on either platform. Fortunately, most of the network applications used in our experiments use a data buffer of 8 KB or larger; however, the FreeBSD FTP program uses a 1 KB data buffer in the client-to-server direction. As we will see in the FTP transfer experiments, this yields lower than expected

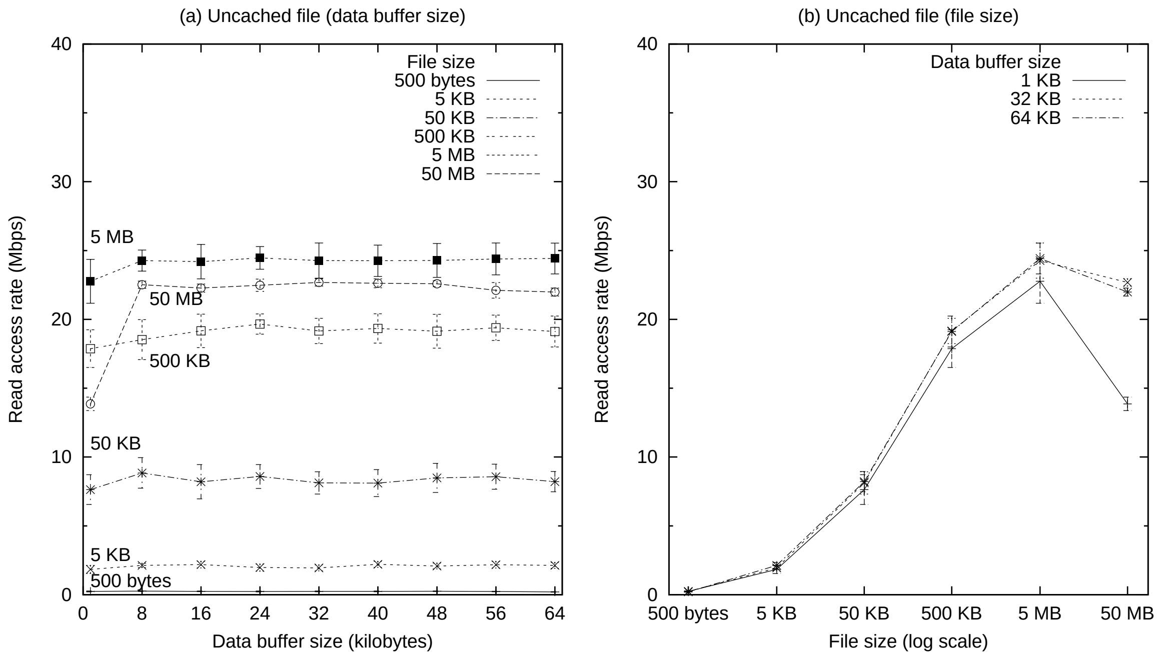


Figure 4.20: File read access rates (Sun, uncached files)

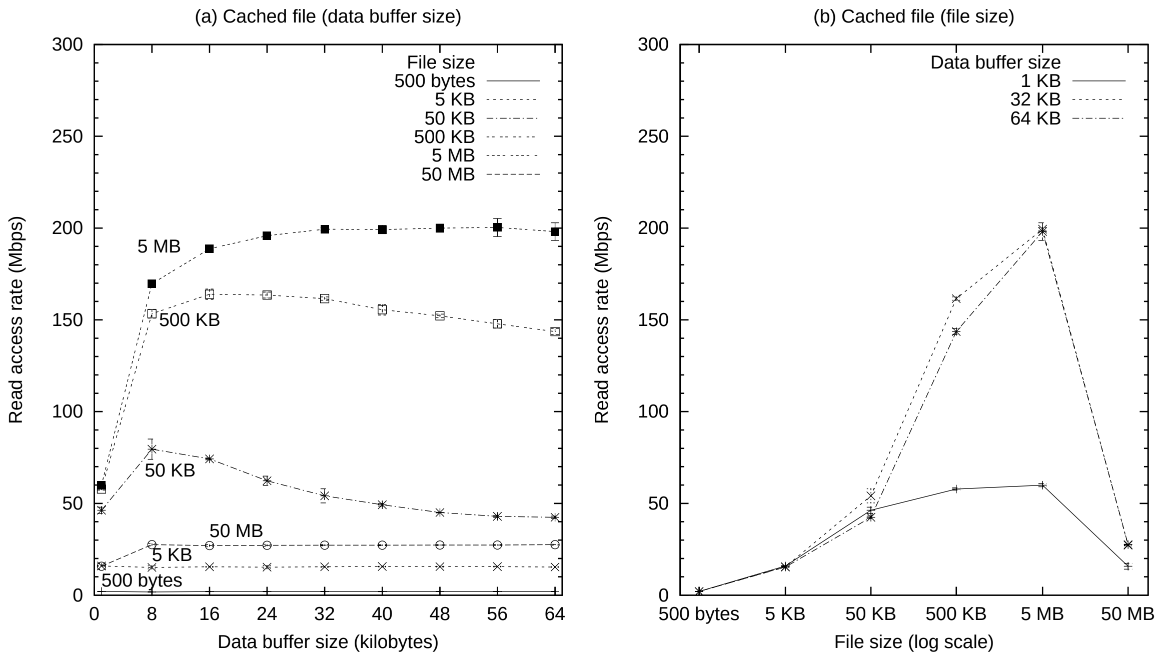


Figure 4.21: File read access rates (Sun, cached files)

throughput.

4.4.3 File Write Access

The SGI platform

Figures 4.22-(a) and (b) show graphs depicting the file write access results obtained on the SGI platform using uncached files. A pattern is observed which is similar to that of the corresponding read access results (Figure 4.18), with the only difference being that the write access rate of each file size is slightly lower than the read access rate. This is likely due to the extra time required for the file block allocations and other file system overhead, which is not required when a file is read from the file system. In this experiment, a maximum rate of 23.0 Mbps is obtained with the 50 MB file size.

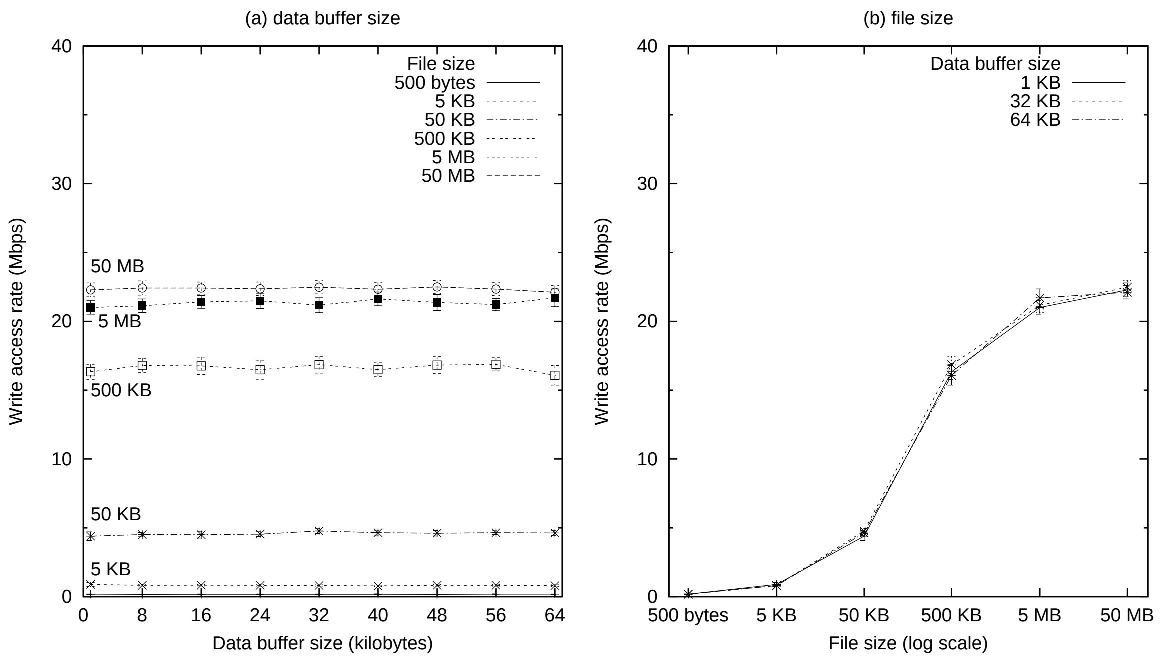


Figure 4.22: File write access rates (SGI, uncached files)

Figures 4.23-(a) and (b) show the results of the file write access experiment in the presence of file caching. The rate improves significantly when compared with those obtained using uncached files. This is because the file is stored in memory instead of on the disk. There is no improvement when using the 50 MB file, indicating that the write file cache is not large enough to store a file of this size, and resulting in the file being written directly to the disk.

A maximum rate of 243.3 Mbps is obtained in this experiment. Note that this value may be slightly higher than the actual rate, as the `write()` system call may not wait for all of the data to be written to the file cache before returning.

The Sun platform

Figures 4.24-(a) and (b) show the results of the file write access experiment on the Sun platform with uncached files. All rates are similar to those in the corresponding read access experiments

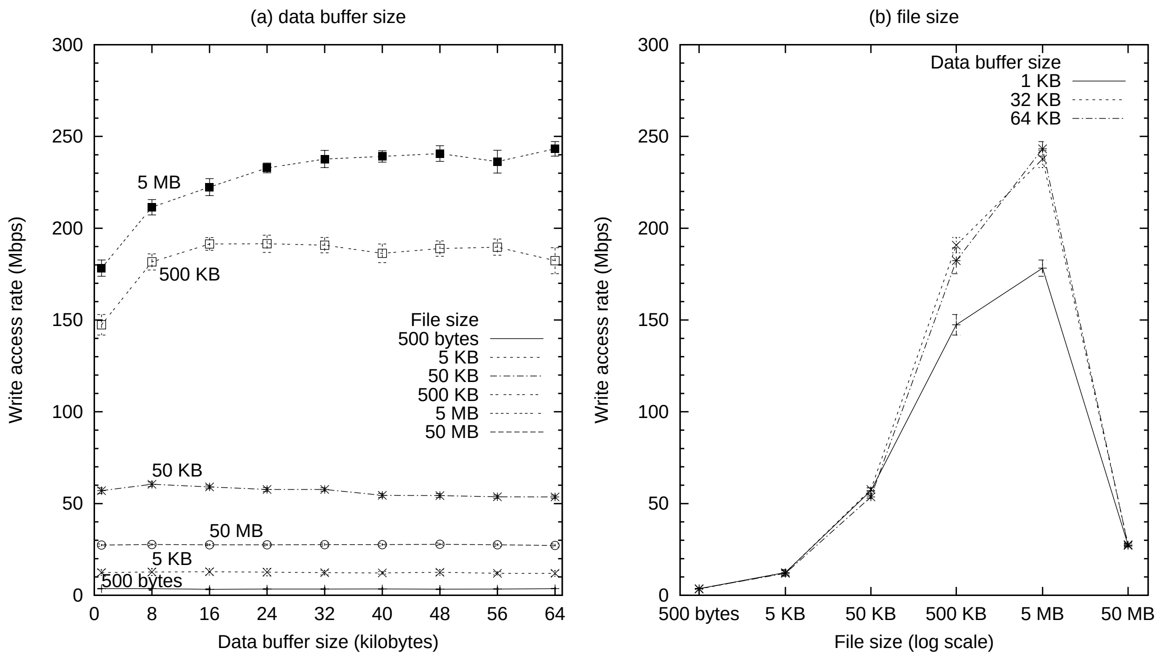


Figure 4.23: File write access rates (SGI, cached files)

(Figure 4.20), with the exception of the 500 KB, which has a lower rate than the others. A maximum rate of 24.2 Mbps is obtained in this experiment.

Figures 4.25-(a) and (b) show the results of the file write access experiment in the presence of the file caching. Improvements are observed when compared with the corresponding access rates of the uncached files, but these improvements are not as significant as those on the SGI's. This is because the default mode for Solaris' file system is to write in a synchronous mode (i.e., data is written immediately to the disk).

Higher rates are observed when compared to those obtained using the uncached files (Figures 4.25), confirming that the `write()` system call does not wait for the data to be written before returning, which results in slightly faster rates. We infer from the results that the 5 MB and 50 MB files are not cached, since the access rates are exactly the same as those of the uncached files. A maximum rate of 39.3 Mbps is obtained using the 50 KB files in the experiment.

4.4.4 Summary of File Access Experiments

The results from the file access experiments are most useful when examining the later HTTP and FTP experiments in which files are transfered. In particular, these results set upper bounds on the transfer rates for applications that perform file access. The maximum access rates obtained from all file access experiments are listed in Table 4.4. Note that the write access rate of 39.3 Mbps obtained using the cached files on the Sun's is lower than expected, because the default mode for Solaris' file system is to write data immediately to the disk.

We observe from all the file access experiments we have conducted that large uncached file sizes (5 MB and 50 MB) generally result in good throughputs (e.g., about 25 Mbps for both read and write on both platforms). A low access rate is obtained on small uncached file sizes (e.g., less than 1 Mbps for the 500 byte file), since the proportion of file system and disk overheads (e.g., seeks

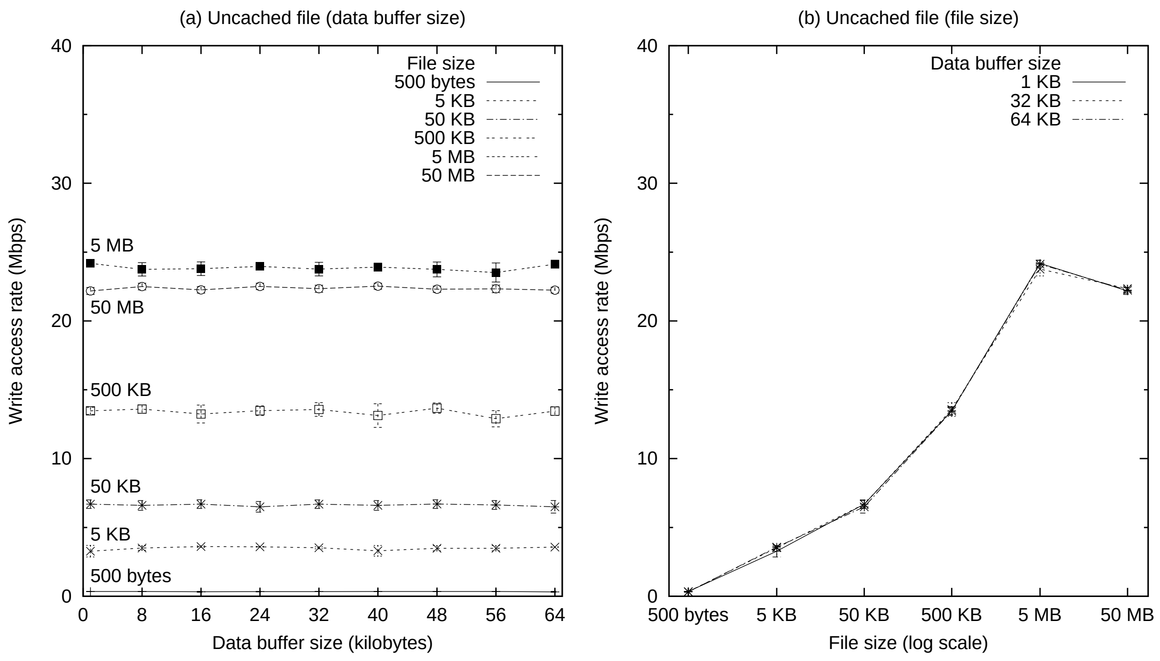


Figure 4.24: File write access rates (Sun, uncached files)

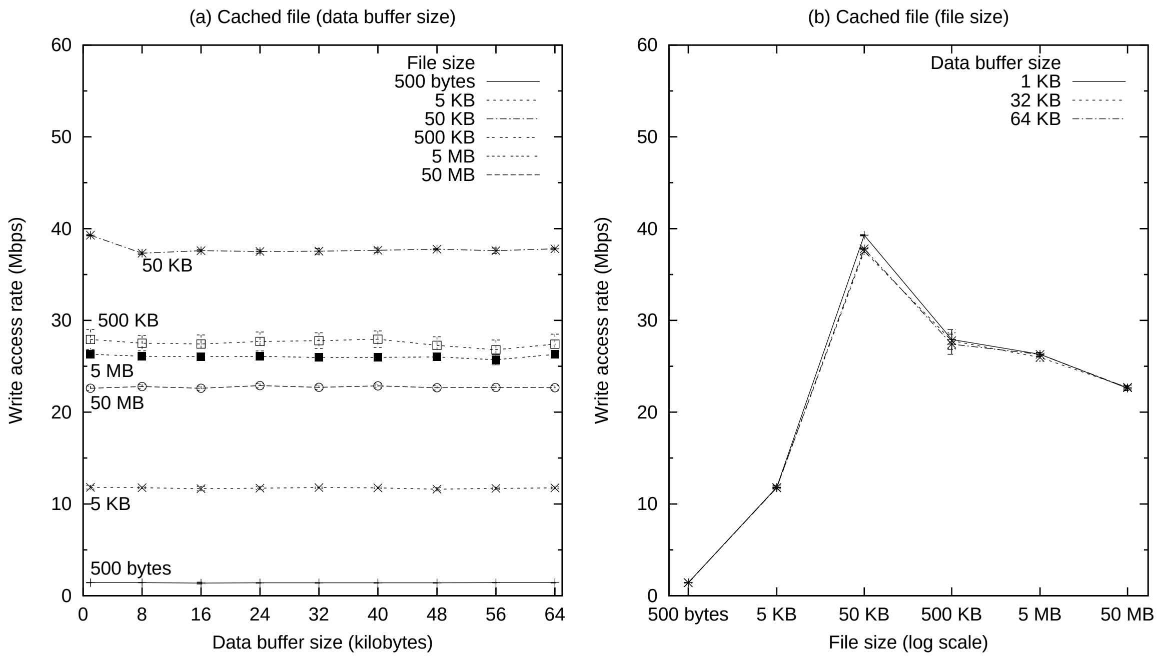


Figure 4.25: File write access rates (Sun, cached files)

	Read access (Mbps)		Write access (Mbps)	
Platform	Uncached file	Cached file	Uncached file	Cached file
SGI	26.8	266.6	23.0	243.3
Sun	24.5	164.0	24.2	39.3

Table 4.4: Maximum file access rates

and rotational latency) is greater for these files. On the other hand, large cached files yield much lower throughputs in comparison with small cached files, because the file cache is not large enough for these files, and this results in the files being directly read from or written to disk.

The impact of the choice of data buffer size is significant when small data buffers are used (less than about 8 KB), but larger data buffer sizes do not affect the access rate in general. With the exception of the FreeBSD FTP program which uses a 1 KB data buffer in the client-to-server direction, all other network applications involved in the later HTTP and FTP experiments use a data buffer of 8 KB or larger. Hence, if we observe any impact of data buffer size on throughput, neither the file systems nor the disks are likely to be a cause.

The experimental results using cached files are much faster than those using the uncached files. This indicates that if appropriate caching schemes are applied to an HTTP or FTP server, its performance is expected to improve significantly.

In the HTTP transfer experiments, file caching is permitted on the HTTP server. Therefore, if the network is not taken into consideration, upper bounds of 266.6 Mbps on the SGI's and 164.0 Mbps on the Sun's are expected. Uncached files are used in the FTP transfer experiments. Thus, theoretical upper bounds of 26.8 Mbps on the SGI's and 24.5 Mbps on the Sun's are expected.

We now evaluate the performance of the HTTP and FTP applications, in order to find out if their upper bounds can be reached, and if not, to determine the main sources of the bottlenecks. We will also compare the throughout obtained using the default parameters of the applications, with those obtained using the optimal parameters we obtained from the previous memory-to-memory transfer experiments.

4.5 HTTP Transfers

This section describes the HTTP transfer experiments in which the performance of the HTTP applications in the three network environments is studied. We measure the throughput on the client side using the application's default parameters, and compare the results with those obtained using the optimal parameters.

4.5.1 Methodology

We have modified the NCSA HTTP daemon (HTTPd) [36] and the HTTP benchmark WebStone [62] to support the native ATM API (see Sections 3.4.4 and 3.5.5 for details). Throughout the experiments in this section, the HTTP daemon is run on the server host and listens to the TCP port 8000 for the Ethernet and TCP/ATM environments, and ASAP 8000 for the native ATM environment.

WebStone provides several metrics to measure the HTTP server's performance. In particular, we are interested in the relationship between the transferred file size and the client throughput. Webstone calculates the client throughput by dividing the total number of bytes received from the server (including the HTTP response header and body) by the total "client response time". The client response time is the time required for the entire connection in which the HTTP request and response are made, measured from just before the `socket()` call (which creates the socket descriptor) through until just after the `close()` call for that socket on the client side. Only successful connections are counted towards the response time. If an error occurs during a request, any bytes read for that request are not counted towards the total.

In our study, two sets of parameters are compared for each network environment on each platform: the default combination of socket buffer, data buffer, and ATM MTU used by the HTTPd daemon and the WebStone benchmark, (we call this combination the "default parameters"), and the optimal combination of these parameters obtained by the memory-to-memory transfer experiments in Section 4.3 (we call this combination the "optimal parameters") . Note that we call the set "optimal" because they yield the optimal throughput in the memory-to-memory transfer experiments, but they may not necessarily yield the optimal throughput with the network applications. The two sets of parameters are shown in Table 4.5. Both the NCSA HTTPd and Webstone applications use an 8 KB data buffer and the default socket buffer assigned by the system. In each experiment, the same set of parameters is used by both applications.

	Socket buffer size (bytes)		Data buffer size (bytes)		ATM MTU (bytes)	
	SGI	Sun	SGI	Sun	SGI	Sun
Default	61440	(default)	8192	8192	9188	9188
Optimal (Ethernet)	8760	65536	8192	4096	N/A	N/A
Optimal (TCP/ATM)	262144	(default)	61440	16240	65535	65535
Optimal (native ATM	N/A	N/A	24576	16308	65535	65535

Table 4.5: Network parameters used in the HTTP transfer experiments

An official set of "standard WebStone 2.0.1 run rules" [55] is provided by Webstone to improve the conformity in WebStone performance results done by different researchers. Under this guideline, a standard file set should be used, which represents the actual load conditions on popular servers (based on a study of file access patterns conducted by SPEC [12]). Each file in the set has a frequency of occurrence, as shown in Table 4.6. Each time a WebStone client makes a request, it randomly selects a particular file to fetch with probability based on the frequency of occurrence. The run rules also impose a duration of at least 10 minutes for each test in the experiment, and the number of clients must vary from 20 to 100 clients, in increments of 10.

File size		Frequency (out of 1000)
500	bytes	350
5	KB	500
50	KB	140
500	KB	9
5	MB	1

Table 4.6: The Webstone standard file set [55]

In our experiments, the five file sizes provided by the official run rules are tested separately, therefore ignoring their frequencies of occurrence. From Table 4.6, it is reasonable to believe that the occurrence of an HTTP request of a 50 MB file is extremely rare in real systems, and this file size (which is used in later FTP experiments) is not used in our HTTP experiments. One client is used to repeatedly fetch one file with a specified size for a period of 10 minutes, and the resulting measurements are averaged (the calculation of confidence intervals is automatically performed by Webstone). We only use one client in each experiment, because at this time we decided not to investigate multiple client scenarios in our study. Since our experiments do not exactly conform to the official "standard WebStone 2.0.1 run rules", their results should not be directly compared with other Webstone benchmark results.

It should be pointed out that conducting experiments with uncached files would produce rather conservative results, since files from the HTTP servers are usually cached in reality. Hence, we make no attempt to flush the file cache on the server. The files in our experiments are therefore expected to be cached on the server, as only one file for each file size is repeatedly fetched. However, we do not rule out the possibility of imperfect caching, especially when large files are used.

The graphs in this section illustrate the experimental results, with the horizontal axis, which is drawn using a log scale, representing the file sizes. The 95% confidence intervals are also shown.

4.5.2 Ethernet

The SGI platform

Figure 4.26-(a) depicts the results obtained using the SGI's and the Ethernet network environment. The throughputs obtained are bounded by the peak throughput of 8.1 Mbps obtained in the memory-to-memory transfer experiment (a line showing this value is also included in the graph). The throughput decreases significantly as the file size decreases (this pattern is also observed in all other HTTP transfer experiments). It is reasonable to believe that the files with sizes as small as 500 bytes and 5 KB are cached during the experiment. As seen in Figure 4.19, the read access rates of all cached file sizes are above 10 Mbps, therefore the limitations imposed by the file system cannot be the major cause of the low throughputs. The inclusion of the HTTP request time in the overall transfer time is likely a major cause, since this time is proportionally large for small files (note that the corresponding FTP experiments do not include the FTP request in the timing).

The use of the optimal socket and data buffer sizes significantly improves throughput for the 500 KB and 5 MB file sizes. A maximum throughput of 7.8 Mbps is observed using these optimal parameters on the largest file (5 MB). Here, an improvement of a factor of 1.2 is obtained over the throughput of 6.3 Mbps obtained using the default parameters.

The Sun platform

The results of the HTTP transfer experiments using the Ethernet environment on the Sun's, presented in Figure 4.26-(b), show that throughput decreases with file size, due to the same reasons as in the previous SGI's experiments.

The optimal parameters yield slightly lower throughputs than the default parameters with the 5 KB and 50 KB files (this is also observed in the corresponding FTP transfer experiment in Figure 4.32-(a)). We have conducted an additional test with the two file sizes, using a 64 KB socket buffer and a data buffer of size 8 KB instead of 4 KB, and obtain throughputs similar to those obtained using the default parameters. This suggests that the lower throughput is mainly caused by the use of the 4 KB data buffer. The fact that the optimal parameters yield better throughput than the default parameters using the 5 MB file supports our choice of the 4 KB data buffer as the optimal data buffer.

The best throughput observed is 8.6 Mbps using the optimal parameters and the 5 MB file. In this case, the throughput is improved by a factor of 1.2 when compared with the 7.7 Mbps obtained using the default parameters.

From the results on both platforms, we observe that the specific combinations of socket and data buffer sizes which yield the optimal throughputs using the Ethernet in the memory-to-memory transfer experiments, also yield good throughputs on large file sizes (e.g., 5 MB in our experiments), but this may not be the case when small files (500 byte, 5 KB, and 50 KB in our experiments) are used. This indicates that throughput behaviour observed in memory-to-memory transfers does not likely reflect the behavior observed in the network applications.

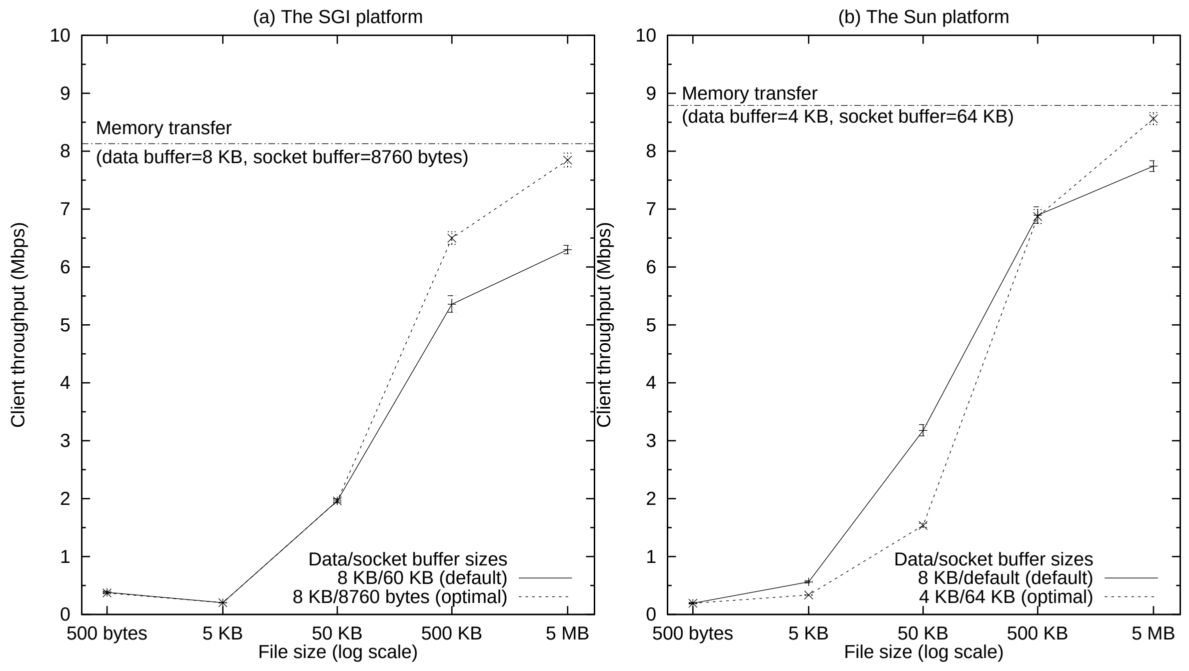


Figure 4.26: HTTP transfers over Ethernet

4.5.3 TCP/ATM

The SGI platform

Figure 4.27-(a) presents the throughputs obtained from the HTTP experiments on the SGI platform using the TCP/ATM environment (the memory-to-memory transfer upper bound of 121.4 Mbps is not shown in the graph because it is out of range).

The use of the default buffer sizes shows low throughputs for the smallest three file sizes (500 byte, 5 KB, and 50 KB), but much higher throughputs are observed for the other two sizes. The optimal parameters show improvements when compared with default parameters when using the 50 KB, 500 KB and 5 MB files, by factors of 7.8, 1.9, and 1.2 respectively. In order to find out if this improvement is due to the ATM MTU size or the optimal buffer sizes, an additional test is conducted using the default buffer sizes (i.e., the default socket and data buffer sizes) and the maximum ATM MTU, and the results are also shown in Figure 4.27-(a). With the exception of the 5 MB file, the resulting throughputs are equal to those obtained using the optimal parameters. This leads us to conclude that the change to the maximum ATM MTU is the dominant factor contributing to the improvement.

The best throughput observed is 48.2 Mbps, obtained using the optimal parameters and the 5 MB file, which improve the throughput by a factor of 1.23 over the default parameters. Note that this is only 39.7% of the memory-to-memory transfer upper bound.

The Sun platform

Figure 4.27-(b) depicts the results obtained from the HTTP experiments using the Sun's and the TCP/ATM environment. It shows very low throughput obtained when using the default parameters. With the exception of the 500 byte and 5 KB files, whose throughputs are expected to be low (as observed in the results using the Ethernet, Figure 4.26-(b)), the other three files yield unexpectedly low throughputs (about 1.3 Mbps each). The use of the maximum ATM MTU shows enormous improvement for the 50 KB, 500 KB, and 5 MB files (improved by a factor of 6.86, 19.77, and 21.33 respectively). This clearly shows that the use of the default ATM MTU is not desirable in this environment. The use of the optimal socket and data buffers further improves the throughput for the 500 KB and 5 MB files by a further factor of 1.14 and 1.46 respectively.

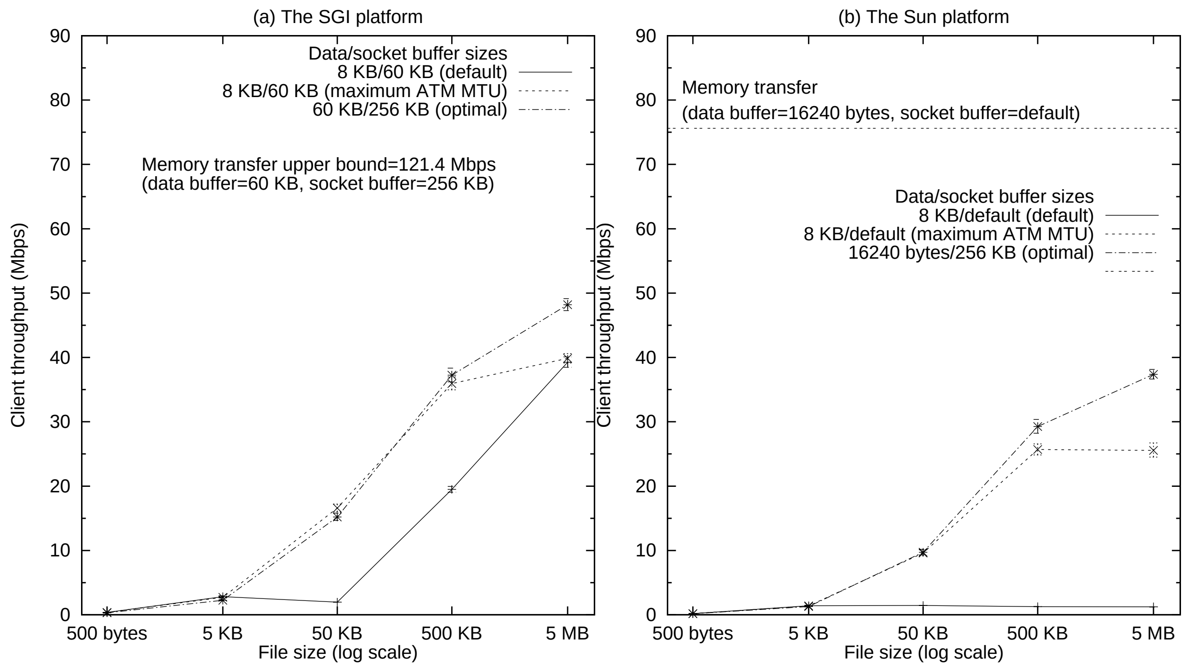


Figure 4.27: HTTP transfers over TCP/ATM

A maximum throughput of 37.4 Mbps is observed with the 5 MB file using the optimal parameters, an increase of a factor of 31.17 from the default buffer and ATM MTU sizes. Note that this is only 49.5% of the memory-to-memory upper bound.

Recall that our previous memory-to-memory transfer experiments, as well as other studies described in Section 2.4.3, show that TCP/ATM significantly out-performs the Ethernet in raw data transfer experiments using simple network benchmarks. Our Webstone results described in this section show that significant improvements are also observed at the application level.

4.5.4 Native ATM

Figure 4.28-(a) compares the results of the HTTP transfer experiments using the default and optimal data buffer sizes on the SGI's in the unreliable native ATM environment. Figure 4.28-(b) compares the corresponding results on the Sun's. Note that we only compare different data buffer sizes here, because we only use one ATM MTU size (the maximum and default ATM MTU's yield the same throughput), and no socket buffers (sockets are not implemented in this environment).

Table 4.7 shows the amount of data loss reported in the experiments on all file sizes tested on both platforms. This is the total number of bytes received as a percentage of the total number of bytes expected. The amounts of data loss when using the default and optimal parameters are shown in the third and fourth columns respectively. From the table, we observe tiny amounts of

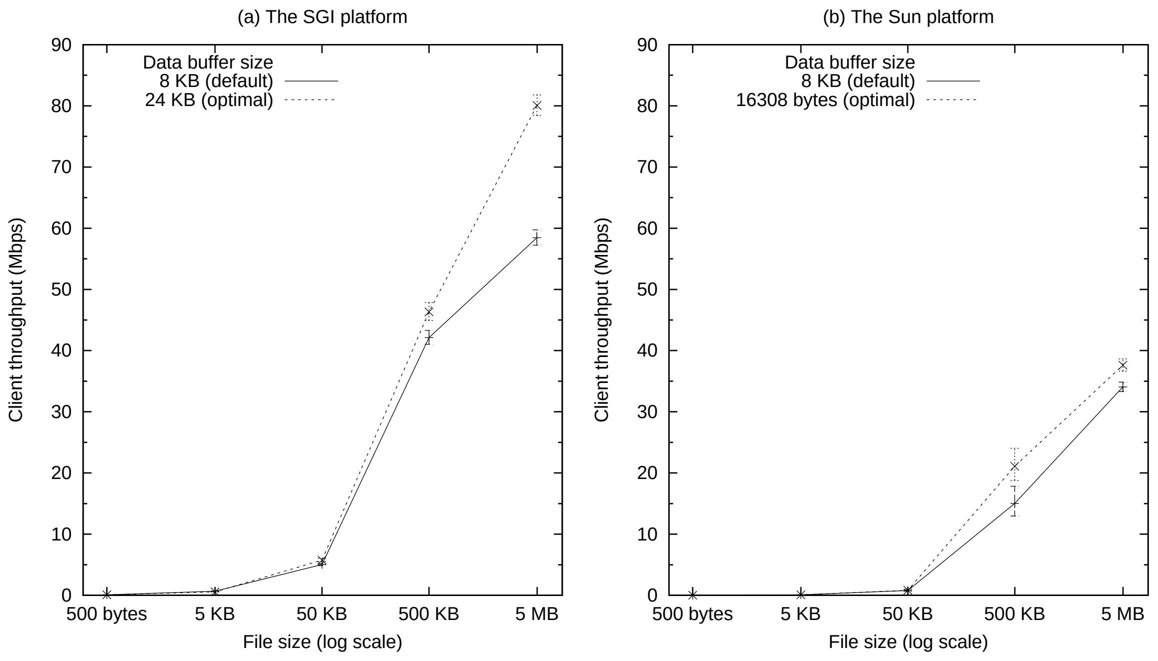


Figure 4.28: HTTP transfers over unreliable native ATM

data loss when using the 500 KB and 5 MB files on both platforms, and none when the other sizes are used.

Platform	File size	Default data buffer size	Optimal data buffer size
SGI	500 bytes	0%	0%
	5 KB	0%	0%
	50 KB	0%	0%
	500 KB	0.15%	0.16%
	5 MB	0.35%	0.28%
Sun	500 bytes	0%	0%
	5 KB	0%	0%
	50 KB	0%	0%
	500 KB	0.20%	0.36%
	5 MB	0.24%	0.23%

Table 4.7: Data loss in HTTP transfers over unreliable native ATM

On both platforms, the optimal data buffer size achieves better throughputs in comparison with the default data buffer size when using the 500 KB and 5 MB files. The maximum throughputs of 80.1 Mbps on the SGI's and 37.6 Mbps on the Sun's are obtained using the 5 MB files, resulting in improvements of factors of 1.37 and 1.10 respectively from those obtained using the default data buffer size. Although these throughputs on the unreliable native ATM environment are not very meaningful (the throughput measurement is based on the amount of data received, which may not be accurate due to possible data loss), they are used as a reference point for determining the amount of performance deterioration caused by the introduction of the reliable protocol in the next set of experiments.

Figure 4.29-(a) shows the results of the HTTP experiment using the SGI's and the reliable native ATM environment. Figure 4.29-(b) shows the corresponding results on the Sun's. An ATM window size of 400 KB is used on both platforms (selected based on the results of the memory-to-memory transfer experiments in Section 4.3.6). The throughputs obtained are close to those obtained when using the unreliable native ATM environment, indicating that no large performance loss is suffered with the presence of our reliable protocol.

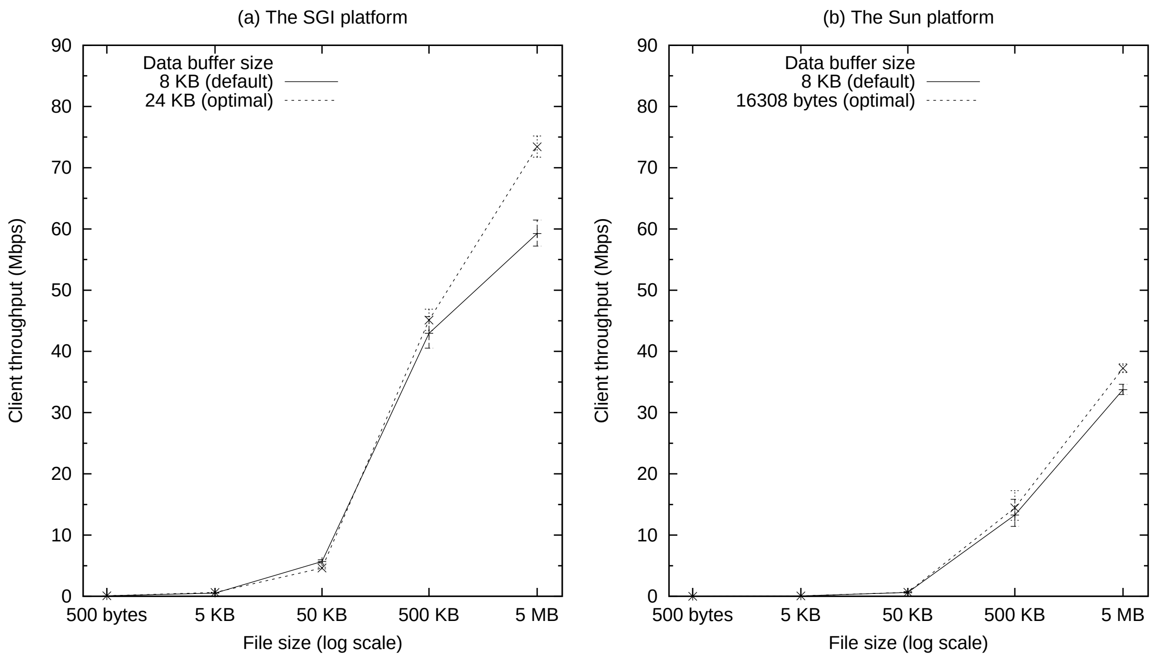


Figure 4.29: Webstone file transfer over reliable native ATM

Retransmissions reported in our experiments for all file sizes are shown in Table 4.8, with those obtained using the default and optimal data buffer sizes shown in the third and fourth columns respectively. Each number is the total number of retransmissions which occurred during the 10 minutes of testing. We only observe retransmissions on the SGI platform when using the 500 KB and 5 MB files, and the use of the optimal data buffer size reduces the number of retransmissions significantly (e.g., from 12 retransmissions to no retransmission when using the 5 MB file). No retransmissions are reported on the Sun's for all file sizes using either data buffer size. This indicates that our reliable protocol does help to prevent data loss and retransmissions on the Sun's (we know from Table 4.7 that a tiny amount of data loss occurs on this platform without our reliable protocol). It is likely that by sending data in sessions, our reliable protocol can help to prevent buffer overflowing, and hence data loss and retransmissions.

The peak throughputs of 73.4 Mbps and 37.2 Mbps are observed on the SGI and Sun platforms, respectively. Both systems obtained these peak throughputs when using the optimal data buffer sizes and the 5 MB files. The throughputs are improved by factors of 1.28 for the SGI's and 1.10 for the Sun's when compared with throughputs obtained by using the default parameters. While the peak throughput on the Sun's is similar to that obtained using the TCP/ATM environment, the peak throughput on the SGI's results in a improvement of a factor 1.52 when compared with the corresponding peak TCP/ATM throughput.

Our previous memory-to-memory transfer experiments, and other studies described in Section 2.4.3, show that native ATM significantly out-performs the Ethernet in raw data transfer experiments using simple network benchmarks. Our results described in this section also show

Platform	File size	Default buffer sizes	Optimal buffer sizes
SGI	500 bytes	0	0
	5 KB	0	0
	50 KB	0	0
	500 KB	13	1
	5 MB	12	0
Sun	500 bytes	0	0
	5 KB	0	0
	50 KB	0	0
	500 KB	0	0
	5 MB	0	0

Table 4.8: Number of retransmissions in HTTP transfers over reliable native ATM

significant improvements at the application level.

4.5.5 Summary of HTTP Transfers

Figure 4.30 summarizes and compares the throughputs achieved in the three network environments on the two platforms using the optimal parameters. The results, when compared with those from the memory-to-memory transfer experiments, clearly show that throughput is significantly lower in real network applications than when executing simple network benchmarks. Results of the HTTP transfer experiments show that throughput decreases as file size decreases, and we expect that this is caused by HTTP protocol overhead (mainly the HTTP request time) which is proportionally larger in small files.

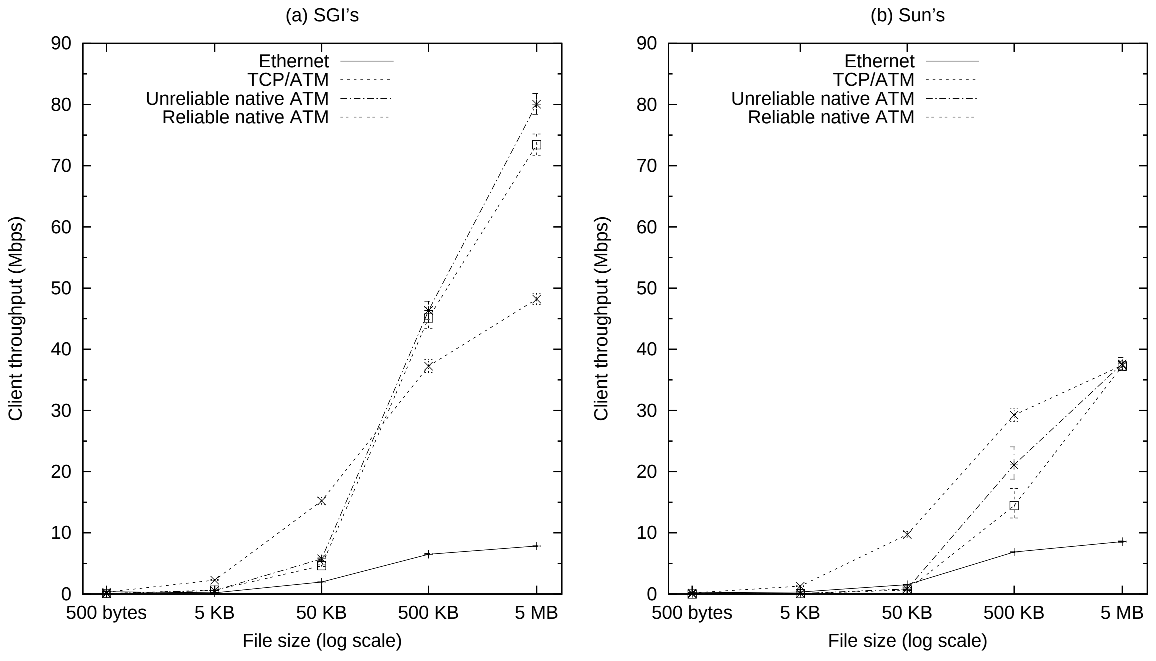


Figure 4.30: Summary of HTTP transfers

The performance of the HTTP applications is similar in the unreliable and reliable native ATM environments, indicating that the reliable native ATM protocol we have implemented does not introduce much overhead when using the ATM network.

The experimental results show that promising throughputs are obtained on large files (500 KB and 5 MB), in comparison with the smaller ones. According to the studies which examine HTTP file sizes in real environments [12] (summarized in Table 4.6), files of sizes 50 KB and 5 MB occur rarely in real systems, and thus these throughputs (over 10 Mbps) are expected to be rarely achieved. However, we should bear in mind that normal Internet users are discouraged from requesting files with large sizes, because of the low bandwidths available in current networks. The average file size of HTTP requests are likely to increase as the network bandwidths increase in the future.

With the exception of the 5 KB and 50 KB files using the Sun's and the Ethernet environment, all throughputs obtained using the optimal parameters are either equal to or better than those obtained using the default parameters. This clearly shows that suitable adjustments of the software configurable parameters can indeed improve the performance of the HTTP applications. Table 4.9 summarizes these improvements with the 5 MB files (this file size yields the peak throughput in all experiments). The first two columns list the platform and network environment being used. The third and fourth columns shows the throughputs obtained using the default and optimal parameters, respectively. The last column shows the "improvement factor", which refers to the ratio of the best throughput obtained using the optimal parameters against that of the default parameters.

	Network environment	Throughput (Mbps) Default parameters	Optimal parameters	Improvement factor
SGI	Ethernet	6.3	7.8	1.24
	TCP/ATM	39.3	48.2	1.23
	Native ATM			
	(unreliable)	58.5	80.1	1.37
	(reliable)	59.2	73.4	1.24
Sun	Ethernet	7.7	8.6	1.12
	TCP/ATM	1.2	37.4	31.17
	Native ATM			
	(unreliable)	34.1	37.6	1.10
	(reliable)	33.8	37.2	1.10

Table 4.9: HTTP transfer improvements using the optimal buffer sizes (5 MB file)

The improvements we have obtained clearly indicate that the HTTP applications perform significantly better in TCP/ATM and reliable native ATM environments than in the conventional Ethernet network environment at the application level (HTTP in this case). For example, on the SGI platform, throughput improves by approximately ten times when using the reliable native ATM environment in comparison with the Ethernet environment with either set of parameters.

On the Sun platform, the peak throughput obtained using the reliable native ATM environment is similar to that using the TCP/ATM environment. On the other hand, the peak throughput obtained on the SGI's using the reliable native ATM environment is improved by a factor of 1.52 in comparison with that obtained in the TCP/ATM environment.

The peak throughputs we have obtained on both platforms are still far below memory-to-memory transfer upper bounds (only 66% on the SGI's and 50.0% on the Sun's). This leaves significant room for future improvement. Since these throughputs are obtained in the presence of the file caching, we believe that the optimization of file system and disk performance is not likely to improve the performance of the HTTP applications significantly (with the exception of file caching optimization). Therefore, other kinds of optimizations, such as those of the network hardware, operating systems, and even the application protocol (i.e., HTTP) are strongly recommended.

Webstone includes the HTTP request time in the overall transfer time, and this request time becomes significant when the transfer time is short. We believe that this is the major source of overhead observed in the low throughput obtained with small files, which have short transfer times. In this case, prolonging a TCP connection to deal with multiple HTTP requests (as proposed in HTTP/1.1 [10]) may not be sufficient for improvement. Reducing the number of HTTP requests should be considered, for example all image files referenced on a page could be requested using one message.

The HTTP transfer experiments allow files to be cached on the server, and reasonable throughputs are obtained in both TCP/ATM and (reliable) native ATM environments. However, in the next series of experiments involving FTP transfers on uncached files, we see that the file system greatly limits the FTP application's performance in these environments.

4.6 FTP Transfer

The experiments carried out to test the performance of the FTP applications are described in this section. Similar to the HTTP transfer experiments, we measure the FTP client side throughput using the default parameters of the applications, and then compare them with those obtained using the optimal parameters.

4.6.1 Methodology

We have modified the Wu-ftpd daemon [66] (Section 3.6.4) and the FreeBSD FTP client (Section 3.7.5) to support the native ATM network environment. Communication between the FTP client and server requires the Internet daemon (inetd) to be running on the server side. In order not to conflict with normal FTP sessions in our system, we use port 2001 for the control connection and port 2000 for the data connection, instead of using the well-known ports.

The same set of files used in the HTTP experiments is used in the FTP experiments. Since it is not unusual to transfer a file with a size much larger than 5 MB via FTP in real systems, we conduct additional tests on a 50 MB file. We also take into consideration that the cache-hit ratio is usually low on an FTP server. Therefore, we use a "cache flush" technique similar to that used in the file access experiments (Section 4.4), so that all files transfered are uncached. It should be pointed out that these results are somewhat pessimistic when compared with the FTP performance obtained in real systems, because files which are commonly requested are likely to be cached in this case.

Two sets of parameters are used to test each of the three network environments on each platform: the default combination of socket buffer, data buffer, and ATM MTU used by the HTTPd daemon and the WebStone benchmark (we call this combination the "default parameters"), and the optimal combination of these parameters obtained by the memory-to-memory transfer experiments in Section 4.3 (we call this combination the "optimal parameters") .

Because different data buffer sizes are used by the FreeBSD FTP and Wu-ftpd programs, and the FreeBSD FTP program also uses different data buffer sizes for sending and receiving data, we have decided to run tests involving transfers in both directions. From the client side, the "server-to-client" transfers are issued using the FTP "get" command, whereas the "client-to-server" transfers are issued using the FTP "put" command. All parameters used in the FTP experiments are summarized in Table 4.10. Note that, although the Fore API does not permit data buffer sizes larger than 32763 bytes to be used, it does allow any data buffer size (up to the limit imposed by the network input function call, e.g., `read()`) for receiving.

	Socket buffer size (bytes)		Data buffer size (bytes)		ATM MTU (bytes)	
	SGI	Sun	SGI	Sun	SGI	Sun
Default (Wu-ftpd)	61440	(default)	4096	1024	9188	9188
Default (FreeBSD FTP)	61440	(default)	32768 (get) 8192 (put)	4096 (get) 1024 (put)	9188	9188
Optimal (Ethernet)	8760	65536	8192	4096	N/A	N/A
Optimal (TCP/ATM)	262144	(default)	61440	16240	65535	65535
Optimal (native ATM)	N/A	N/A	24576	16308	65535	65535

Table 4.10: Network parameters used in the FTP transfer experiments

Each run in an experiment consists of an FTP session in which the client connects to and logs on to the server, performs a "get" or "put" with a file of a particular size, and then closes the session. The file is transferred from the sender's `/var/tmp` local directory to the receiver's `/var/tmp` local directory, thus bypassing the network file systems. The client's throughput is measured by timing the duration of the data connection for each transaction (including the connection setup and disconnection times), as shown in Figure 4.2. The control connection is not considered in the measurements, since we have not modified it from its original TCP connection even in the native ATM implementation. Again, we present results which are an average of 20 runs. The graphs in this section use a log scale for the horizontal axis (file size) and also include 95% confidence intervals.

4.6.2 Ethernet

The SGI platform

The throughput results obtained by the FTP experiments using the SGI's and the Ethernet are shown in Figure 4.31-(a) and Figure 4.31-(b) in the "get" and "put" directions respectively. The throughput decreases significantly as the file size decreases, which is likely caused by the file system limitations described in Section 4.4.

The throughputs obtained from the two transfer directions (`get` and `put`) are slightly different. This is not surprising when using the default parameters since the FTP client uses different data buffer sizes in each of the two directions, as shown in Table 4.10. However, the same socket and data buffer sizes are used by both the client and the server for the optimal parameters, and different throughputs are still obtained. We have examined the source code for the server and client programs, and it is unlikely that the exact cause of this difference is the difference in the applications network handling implementations. We believe that the difference may be caused by the network implementation at the operating system level. For example, we observed from Figures 2.9 and 2.10 that the server waits for about 200 ms before receiving data, which does not happen in the opposite transfer direction, resulting in different throughputs being obtained in either of the two directions. We also observe a small difference in throughputs in the rest of the FTP experiments when using the optimal parameters in the two directions.

The throughputs obtained when using the optimal parameters show noticeable improvement when compared with those obtained when using default parameters for the large files (500 KB, 5 MB, and 50 MB). The peak throughput of 7.9 Mbps is achieved in either direction using the 50 MB file. This is close to the corresponding throughput of 8.1 Mbps obtained in the memory-to-memory transfer experiment (Figure 4.3). This is an improvement of a factor of 1.22 from the 6.5 Mbps obtained using the default parameters.

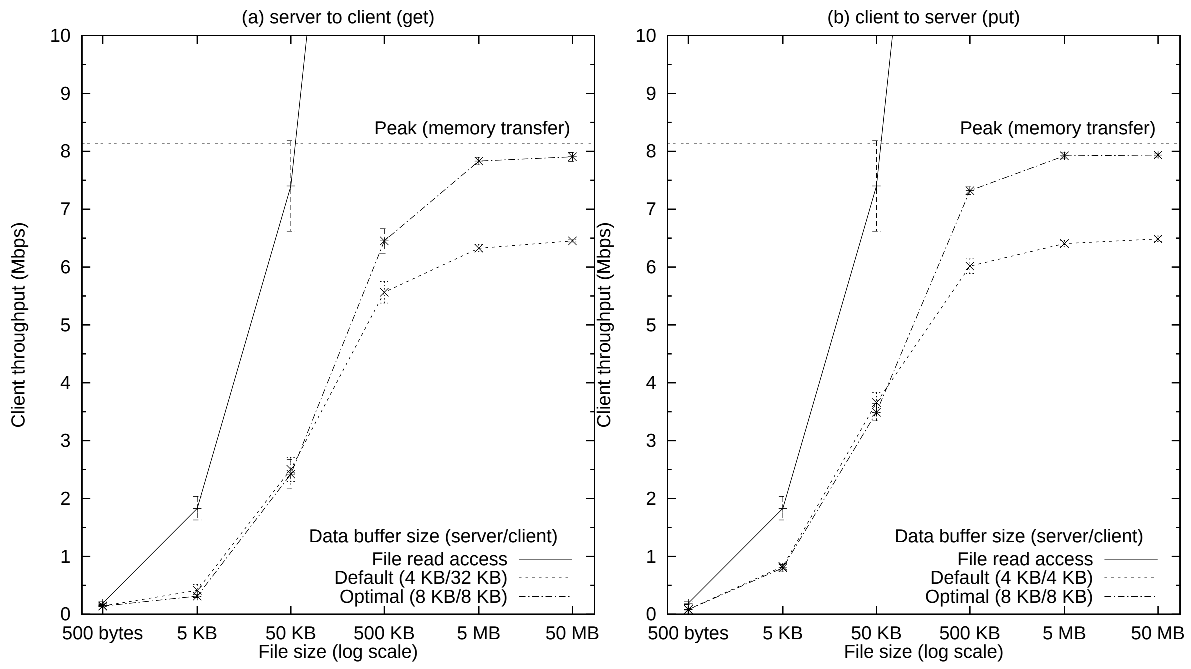


Figure 4.31: FTP transfers over Ethernet (SGI)

The Sun platform

The results obtained by the FTP experiments on the Sun's over the Ethernet, presented in Figure 4.32, show similar behaviour to the SGI's results. In the server-to-client direction, the optimal parameters achieve higher throughput than the default parameters only for the large files (5 MB and 50 MB) but slightly lower throughputs than the default parameters for the 5 KB and 50 KB files (recall that this same pattern is observed in the corresponding HTTP transfer experiment, as shown in Figure 4.26-(b)). As described in Section 4.5.2, we expect that this is caused by the use of the 4 KB data buffer. In the client-to-server direction, however, we are able to obtain higher throughput on all file sizes when using the optimal parameters in comparison with the default parameters.

Peak throughputs of 8.6 Mbps and 8.4 Mbps are achieved using the 50 MB file in the "get" and "put" directions respectively, and they are close to that obtained by the corresponding memory-to-memory transfer experiment (8.8 Mbps in Figure 4.6). When compared with the peak throughputs obtained using the default parameter, these peak throughputs show an improvement factor of 1.08 and 1.12 in the "get" and "put" directions, respectively.

From the above experiments on both platforms, we observe that the bandwidth available on the Ethernet network is a major limitation to the overall performance of the FTP applications.

4.6.3 TCP/ATM

In the TCP/ATM experiments, in order to observe the effect of the ATM MTU on throughput, we conduct tests using the default socket and data buffers along with the maximum ATM MTU, in addition to the other tests.

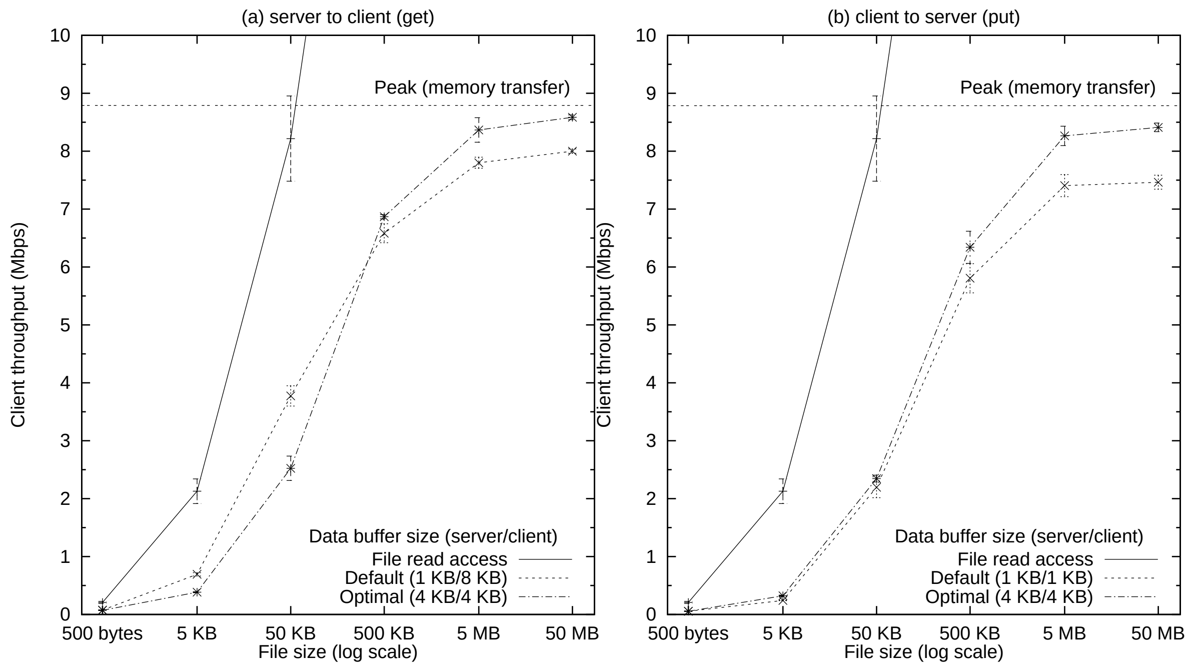


Figure 4.32: FTP transfers over Ethernet (Sun)

The SGI platform

Results from the experiments conducted using the SGI's and the TCP/ATM environment in the "get" and "put" directions are presented in Figure 4.33-(a) and Figure 4.33-(b) respectively. The throughput in each experiment is bounded by the file read access upper bound, which is also shown in the graphs. The throughput decreases significantly with file size, and we believe the file system limitation is the major cause of this relatively low performance (the memory-to-memory transfer upper bound of 121.4 Mbps is much higher than the peak throughput obtained here).

When compared with the default parameters, the throughput obtained using the optimal parameters yield better throughputs with most file sizes, and they are very similar to those obtained when using the the default buffer sizes and the maximum ATM MTU. We therefore conclude that the improvement in throughput is mainly due to the use of the maximum ATM MTU size.

The peak throughputs of 26.0 Mbps and 25.4 Mbps are achieved using the optimal parameters and the 5 MB file size in the "get" and "put" directions, respectively.

The Sun platform

Figure 4.34-(a) presents the results obtained by the FTP transfer experiment on the Sun's in the "get" direction. The optimal parameters out-perform the default parameters on all file sizes. Throughputs obtained using these optimal parameters are statistically similar to those of the default parameters combined with the maximum ATM MTU. The peak throughput of 22.2 Mbps is observed with the 5 MB file size.

Figure 4.34-(b) presents the results obtained in the "put" direction. Poor throughputs are observed using the default parameters. Changing the default ATM MTU to the maximum ATM MTU shows some improvement. Using the optimal parameters gives the best results, yielding a peak throughput of 21.7 Mbps with the 5 MB file size.

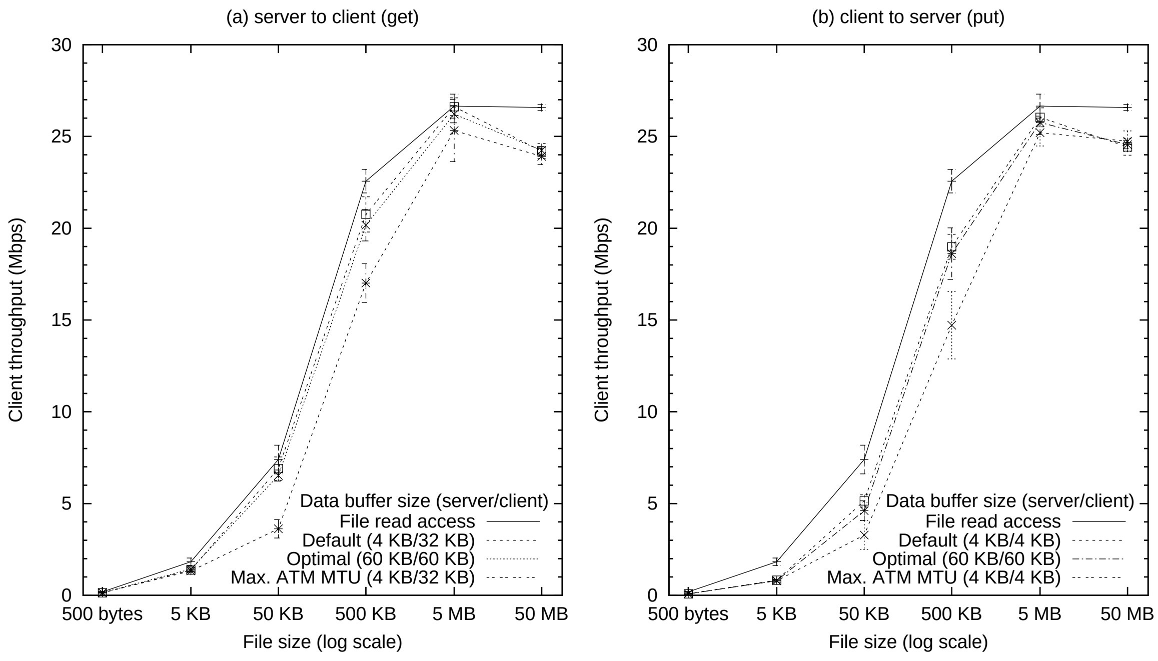


Figure 4.33: FTP transfers over TCP/ATM (SGI)

It is apparent that the default ATM MTU size is a factor in causing the poor performance, since changing it to the maximum size shows significant improvement. The use of the rather small 1 KB data buffer is also a possible factor, as it yields low throughput in the file access experiment (Figure 4.18-(a)), and in the corresponding experiments using the native ATM environment (Figures 4.37 and 4.38), changing the default data buffer size to a larger one also yields improved throughput.

Despite the limitation imposed by the file systems, the FTP applications show significant improvement using the TCP/ATM environment in comparison with the Ethernet environment at the application level (note that our previous experiments only show this achievement in memory-to-memory transfer using simple network benchmarks).

4.6.4 Native ATM

In the native ATM environment, the default and optimal parameters refer only to the data buffer sizes, because we only use one ATM MTU (the maximum ATM MTU), and no socket buffers (sockets are not implemented in this environment).

Note that FTP throughputs obtained using an unreliable transfer environment are rather meaningless for representing the FTP performance. However, these throughputs are used as references when analyzing the throughputs obtained using the reliable native ATM environment.

The SGI platform

Figure 4.35-(a) and Figure 4.35-(b) present the results obtained by the FTP experiments using the unreliable ATM network environments on the SGI's in the "get" and "put" directions, respectively. Figure 4.36-(a) and Figure 4.36-(b) show the corresponding results in the reliable native

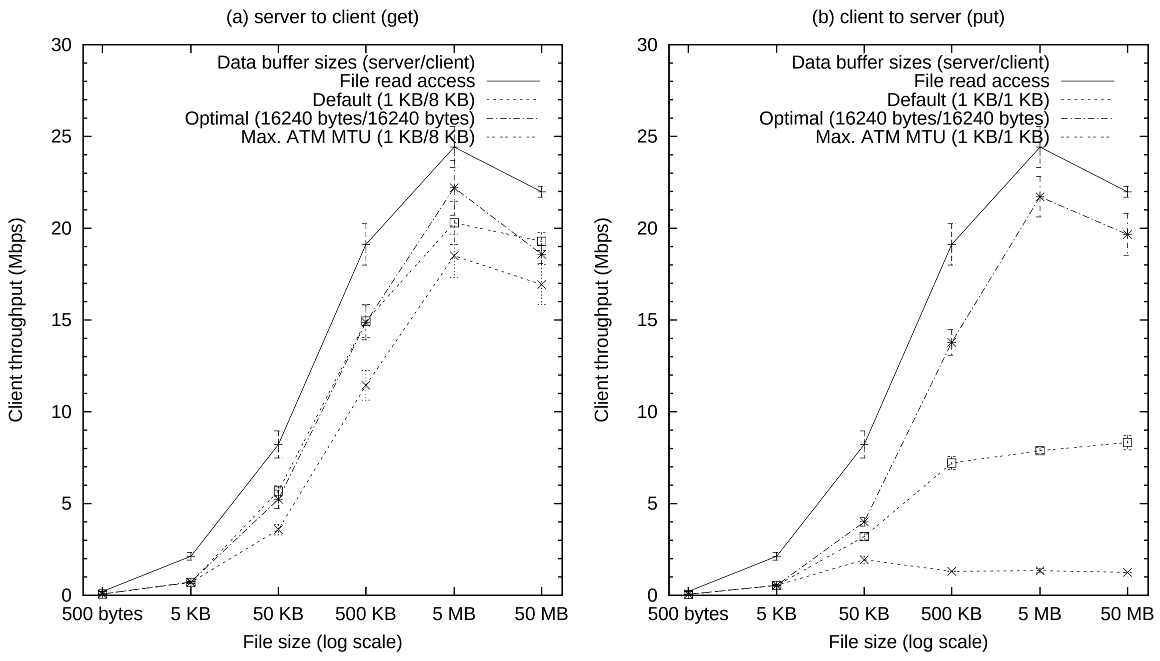


Figure 4.34: FTP transfers over TCP/ATM (Sun)

ATM environments. An ATM window size of 400 KB (selected based on the results of the memory-to-memory transfer experiments in Section 4.3.6) is used in the reliable native ATM experiments. In all tests, the default and optimal data buffer sizes yield very similar throughputs, and all are bounded by the file read access limit (also shown in the graphs).

Table 4.11 shows the amount of data loss that occurs in the unreliable native ATM experiments on both platforms. The second column lists the file sizes being tested. The third and fourth columns show the amount of data loss reported in the "get" direction when using the default and optimal data buffer sizes respectively, whereas the fifth and sixth columns show the amount of data loss in the "put" direction when using the default and optimal data buffer sizes respectively. The amount of data loss is calculated as the total number of bytes received as a percentage of the expected total number of bytes (i.e., the file size).

		Server to client (get) Data buffer sizes		Client to server (put) Data buffer sizes	
Platform	File size	(default)	(optimal)	(default)	(optimal)
SGI	5 KB	0%	0%	0%	0%
	50 KB	0%	0%	0%	0%
	500 KB	0%	0%	0.08%	0%
	5 MB	0.02%	0%	0.03%	0%
	50 MB	11.23%	12.25%	11.66%	10.94%
Sun	5 KB	0%	0%	0%	0%
	50 KB	0%	0%	0%	0%
	500 KB	60.00%	5.53%	56.82%	4.80%
	5 MB	68.25%	1.51%	66.32%	1.70%
	50 MB	59.71%	0.16%	56.29%	0.20%

Table 4.11: Data loss in FTP transfers over unreliable native ATM

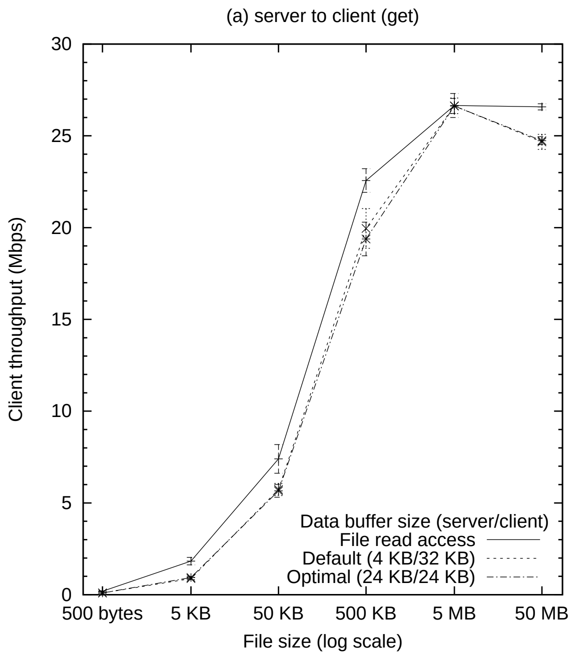
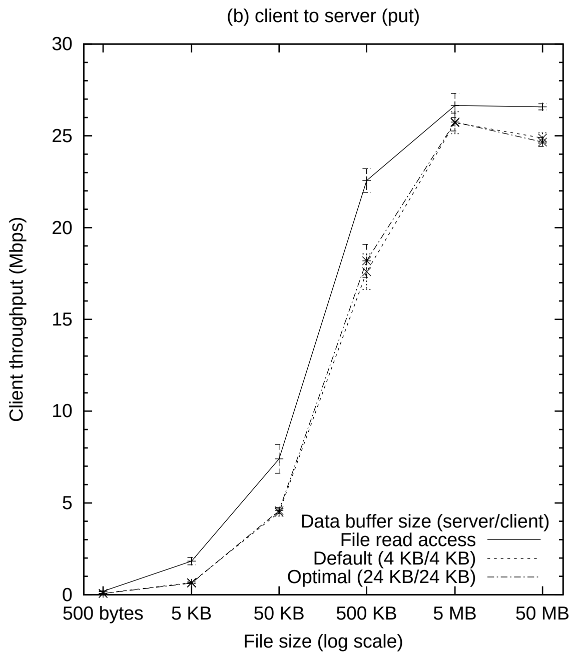


Figure 4.35: FTP transfers over unreliable native ATM (SGI)

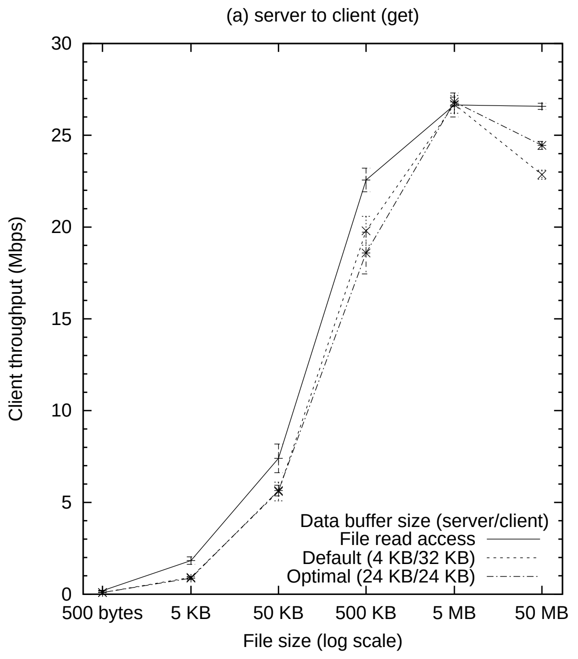
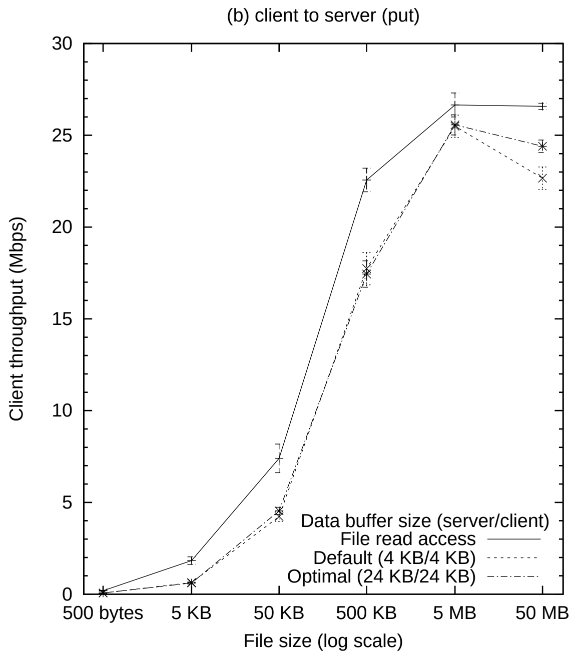


Figure 4.36: FTP transfers over reliable native ATM (SGI)

Table 4.12 lists the number of retransmissions reported in the reliable native ATM experiments on both platforms. Each value is the total number of retransmissions out of the 20 runs. The second column lists the file sizes being tested. The third and fourth columns show the number of retransmissions reported when using the default and optimal data buffer sizes respectively in the "get" direction, whereas the fifth and sixth columns show the number of retransmissions reported when using the default and optimal data buffer sizes respectively in the "put" direction.

		Server to client		Client to server	
		Data buffer sizes		Data buffer sizes	
Platform	File size	(default)	(optimal)	(default)	(optimal)
SGI	5 KB	0	0	0	0
	50 KB	0	0	0	0
	500 KB	0	0	0	0
	5 MB	0	0	0	0
	50 MB	19	0	20	0
Sun	5 KB	0	0	0	0
	50 KB	0	0	0	0
	500 KB	20	0	20	0
	5 MB	21	0	22	0
	50 MB	29	0	33	0

Table 4.12: Number of retransmissions in FTP transfers over reliable native ATM

On the SGI's, the default data buffer sizes result in a tiny amount of data loss when using the 5 MB file size (0.02%), but a more severe loss of 11.23% of the data when using the 50 MB file. The optimal data buffer size also results in a similar amount of data loss with the 50 MB file size, but no data loss for the 5 MB file size. This, together with the fact that no data loss is observed when using the other smaller files, suggests that larger files increase the chance of data loss. We believe that the overflow of memory buffers is likely the cause.

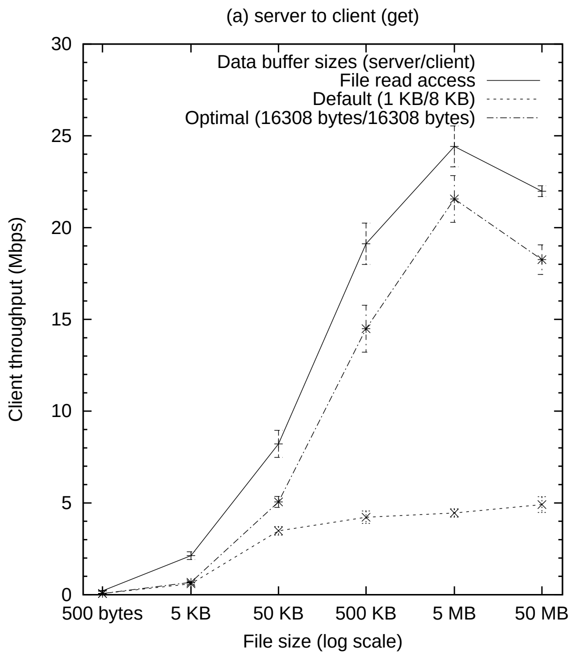
In the corresponding reliable native ATM experiment, a number of retransmissions is reported in both transfer directions when using the 50 MB file size with the default data buffer size, resulting in lower throughputs which are observed in the graphs. On the contrary, there are no retransmissions reported when using the optimal data buffer size, even though data loss of 11% was observed in the corresponding unreliable native ATM experiment. The absence of any retransmissions must be caused by the use of the 24 KB data buffer on the receiver side, as opposed to the default size of 4 KB (since other parameters are unchanged). As we believe that memory buffer overflow is the main cause of data loss in our native ATM environment, a larger data buffer size should ease the network traffic, and therefore lower the chance of any buffer overflow in the network.

Using the reliable native ATM environment, the peak throughputs of 26.8 Mbps and 25.6 Mbps are achieved with the 5 MB file size in the "get" and "put" directions respectively, with no significant improvement compared with those obtained using the default buffer sizes.

The Sun platform

Figure 4.37-(a) and Figure 4.37-(b) present the results obtained by the FTP transfer experiments on the Sun's using the unreliable ATM network environments in the "get" and "put" directions respectively. Figure 4.38-(a) and Figure 4.38-(b) show the corresponding results using the reliable native ATM environments. Again, an ATM window size of 400 KB is used in the reliable native ATM experiment.

Due to the severe loss of data when using the large files (500 KB, 5 KB, and 50 MB) (as shown in Table 4.11), the default data buffer size yields lower throughput with these files in comparison



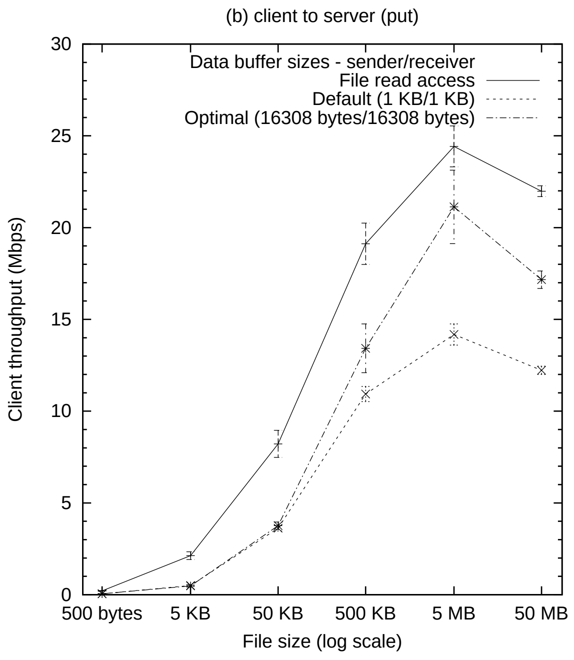
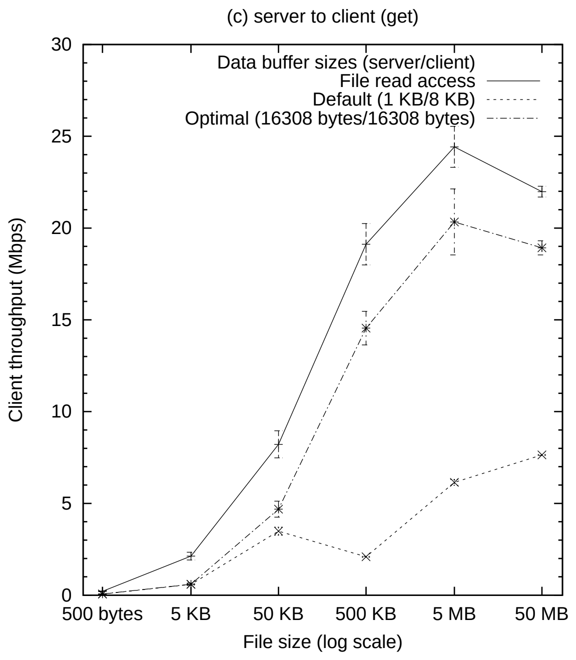


Figure 4.37: FTP transfers over unreliable native ATM (Sun)



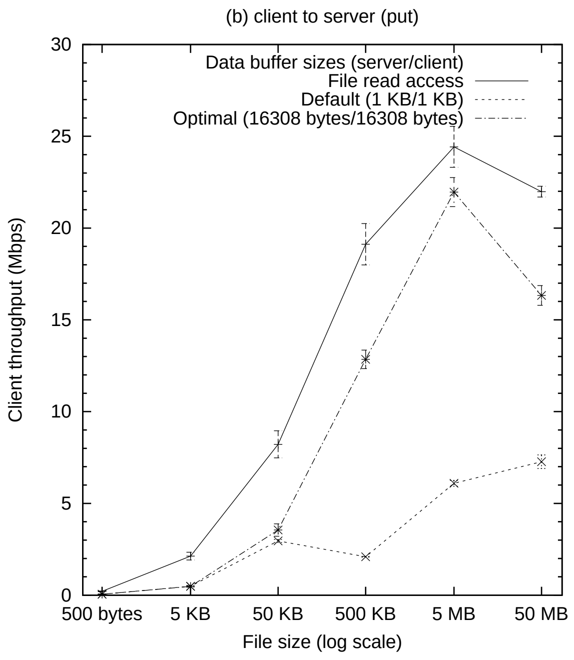


Figure 4.38: FTP transfers over reliable native ATM (Sun)

with the optimal data buffer size. We believe that this is caused by the use of the rather small 1 KB data buffer size, since data loss is greatly reduced by changing the data buffer size to the much larger size of 16308 bytes.

When using the reliable protocol, the lower throughput obtained as a result of the retransmissions can be seen in Figures 4.38-(a) and 4.38-(b) with the default data buffer size. As implied in the corresponding unreliable experiments, the retransmissions account for about 60% of the data being lost (as shown in Table 4.12). By using the optimal data buffer size, we are able to successfully prevent all retransmissions. With the use of the 5 MB file size, the peak throughputs of 20.3 Mbps and 22.0 Mbps are obtained in the "get" and "put" directions respectively. These throughputs are higher than the corresponding throughputs obtained using the default data buffer size by factors of 2.67 and 3.01 in the "get" and "put" directions respectively.

From the results on the SGI and Sun platforms, we observe the significant difference in throughput, amount of data loss, and the number of retransmissions which results with the file sizes tested on the two platforms. We also observe that on the Sun's, the optimal data buffer size successfully prevents all the retransmissions that would occur if the default data buffer size is used instead.

Recall that our previous memory-to-memory transfer experiments, as well as other studies described in Section 2.4.3, show that native ATM significantly out-performs the Ethernet in raw data transfer experiments using simple network benchmarks. The results from our FTP experiments show that the same improvements are observed at the application level.

4.6.5 Summary of FTP Transfers

A summary of the FTP transfer results, showing only the throughputs obtained using the optimal parameters, is presented in Figure 4.39, in which (a) and (b) are the results in the "get" and "put" directions respectively on the SGI's, whereas (c) and (d) are the results in the "get" and "put" directions respectively on the Sun's.

As can be seen in these graphs, throughputs increase with file size in both transfer directions on both platforms. We believe that this is caused by the file system limitation as described in Section 4.4. The throughput when using the Ethernet is bounded by the network bandwidth, whereas throughput obtained when using the TCP/ATM and native ATM environments is bounded by the file access rate imposed by the file systems and disks. As a result, the maximum throughputs of the TCP/ATM and native ATM environments are greatly restricted. For example in the "get" transfer direction, throughputs of only 26.0 Mbps on the SGI's and 22.2 Mbps on the Sun's are obtained using the TCP/ATM environment, and only 26.8 Mbps on the SGI's and 20.3 Mbps on the Sun's are obtained using the reliable native ATM environment. The memory-to-memory transfer upper bounds, as shown in Table 4.2, are much higher than these values.

With the exception of the 5 KB and 50 KB files using the Sun's and the Ethernet, the optimal parameters perform as well as or better than the default parameters on either platform when using each of the three network environments. The optimal parameters are also shown to reduce retransmissions with the reliable native ATM protocol we have implemented.

On the Sun platform, a 1 KB data buffer is used in the original FreeBSD client in the client-to-server direction, because it is the "fragment size" of the machine's local file system. While the resulting throughput is adequate in conventional networks such as Ethernet, it becomes obviously inadequate when the application is executed over a faster network, as it is shown to perform badly in the TCP/ATM and native ATM environments. The optimal parameters we have chosen eliminate this shortcoming.

Table 4.13 compares the throughputs obtained when using both the default and optimal sets of

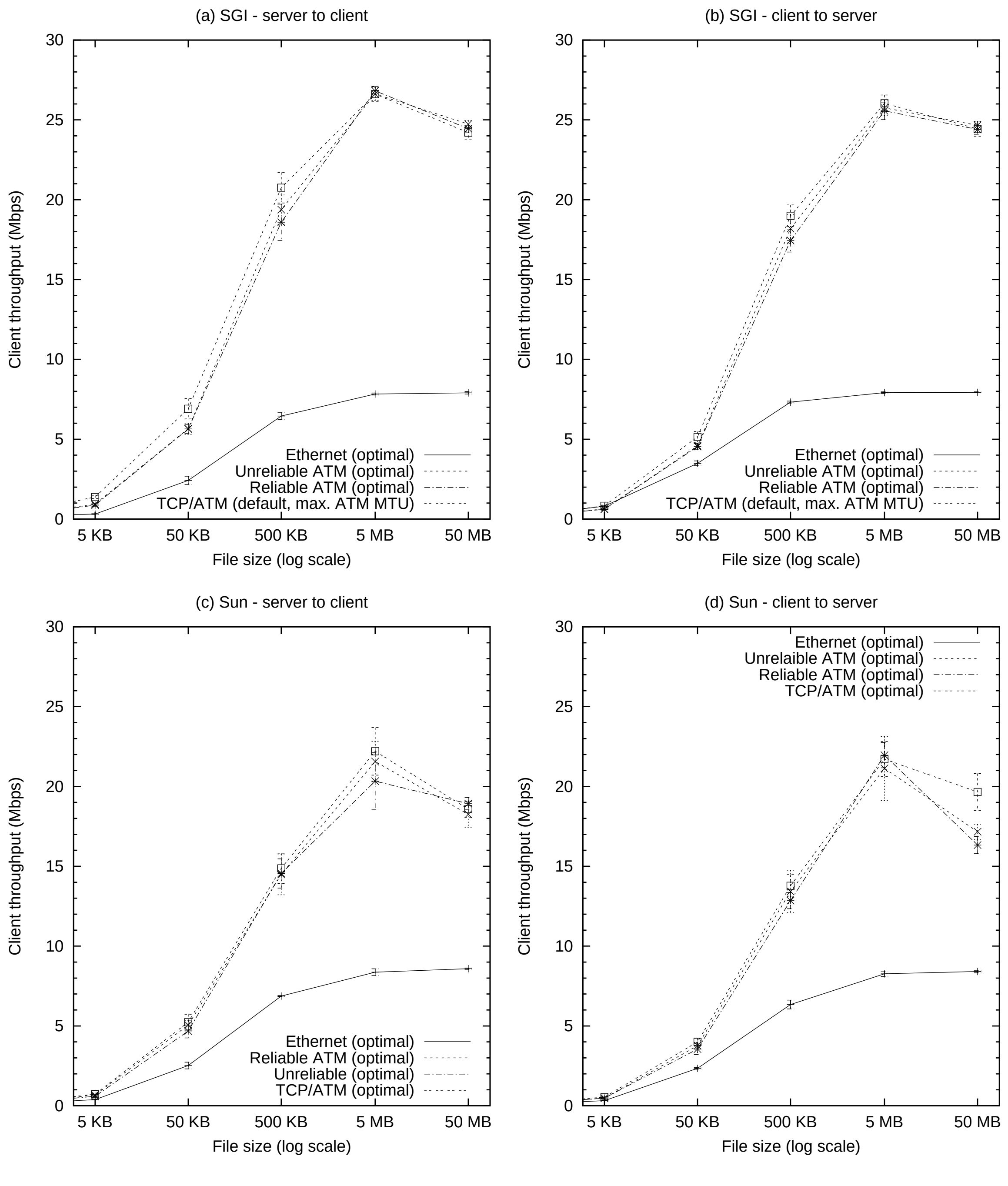


Figure 4.39: Summary of FTP transfers

parameters in all FTP transfer experiments. The first two columns list the platform and network environment being use. The third column specifies the transfer direction, the fourth and fifth columns show the throughputs obtained using the default and optimal parameters respectively. The last column shows the "improvement factor", which refers to the ratio of the best throughput obtained using the optimal parameters against that of the default parameters.

	Network environment	Transfer direction	Throughput (Mbps) Default buffer sizes	Optimal buffer sizes	Improvement factor
SGI	Ethernet	get	6.5	7.9	1.22
		put	6.5	7.9	1.22
	TCP/ATM	get	25.3	26.0	1.03
		put	25.2	25.4	1.01
	Native ATM				
	(unreliable)	get	26.6	26.6	1.00
		put	25.7	25.7	1.00
	(reliable)	get	26.7	26.8	1.00
		put	25.5	25.6	1.00
Sun	Ethernet	get	8.0	8.6	1.08
		put	7.5	8.4	1.12
	TCP/ATM	get	18.5	22.2	1.20
		put	1.9	21.7	11.42
	Native ATM				
	(unreliable)	get	4.9	21.6	4.41
		put	14.2	21.1	1.49
	(reliable)	get	7.6	20.3	2.67
		put	7.3	22.0	3.01

Table 4.13: FTP transfer improvements using the optimal buffer sizes (50 MB file)

Significant improvements are observed when using the Ethernet network and our modified versions of the FTP applications. However, the file read/write access upper bounds limit the FTP performance when using the TCP/ATM and (reliable) native ATM environments, and this results in no improvement on the SGI's. Improvements are observed on the Sun's because the default parameters perform badly with the small 1 KB data buffer used on the FTP client. This suggests that the default parameters of network applications could produce undesirable results in real systems. Suitable tuning of these parameters is therefore necessary.

The peak throughputs we have obtained on both platforms are still far below memory-to-memory transfer upper bounds. This leaves potentially large room for future improvement. Since these throughputs are greatly limited by the file access rate, we believe that the optimization of file system and disk performance of future systems can significantly improve the performance of FTP applications. However, recall from the HTTP experiments that, the presence of the file caching may still yield low throughput. Other kinds of optimizations, such as those of the network hardware, operating systems, and even the application protocol (FTP in this case) are also recommended. All these bottlenecks will become noticeable when networks with higher speeds than the conventional Ethernet networks (such as the ATM networks) become common.

4.7 Summary

In this chapter, we have presented the results and analysis of the memory-to-memory transfer and file access experiments, as well as those of the HTTP and FTP applications.

Figures 4.40 and 4.41 summarize the best throughputs we obtained for all experiments. For each network environment, we show three pairs of throughputs: the first pair is from the memory-to-memory transfer experiments, the second pair from the HTTP transfer experiments, and the last from the FTP transfer experiments. Each pair consists of the throughput obtained using the default and the optimal parameters.

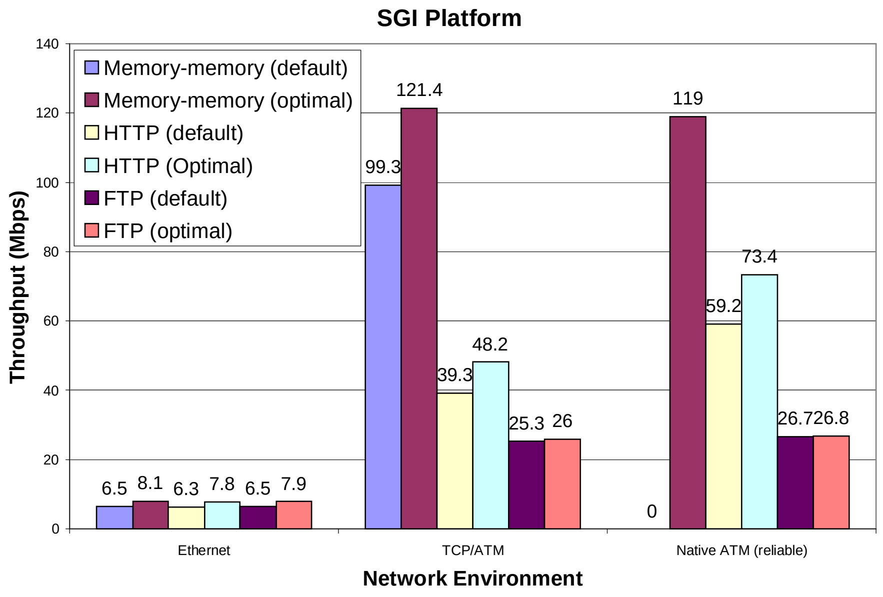


Figure 4.40: Summary of experimental results (SGI)

The optimal parameters we have selected to use in the network applications generally outperform the default parameters, suggesting that suitable tuning of software configurable parameters is important in the optimization of the network applications in real systems.

We have identified some limitations and bottlenecks that prevent our applications from obtaining higher performance. These include the network bandwidth (in the case of the Ethernet), protocol overhead (HTTP request time in the case of the HTTP experiments), and the file system limitation (in the case of the FTP experiments). Therefore, we recommend that optimizations be performed in these areas to improve the performance of the network applications, so that they can fully utilize the high-speed networks of the future.

In the next chapter, we summarize our contributions, present our conclusions, and discuss possibilities for future work.

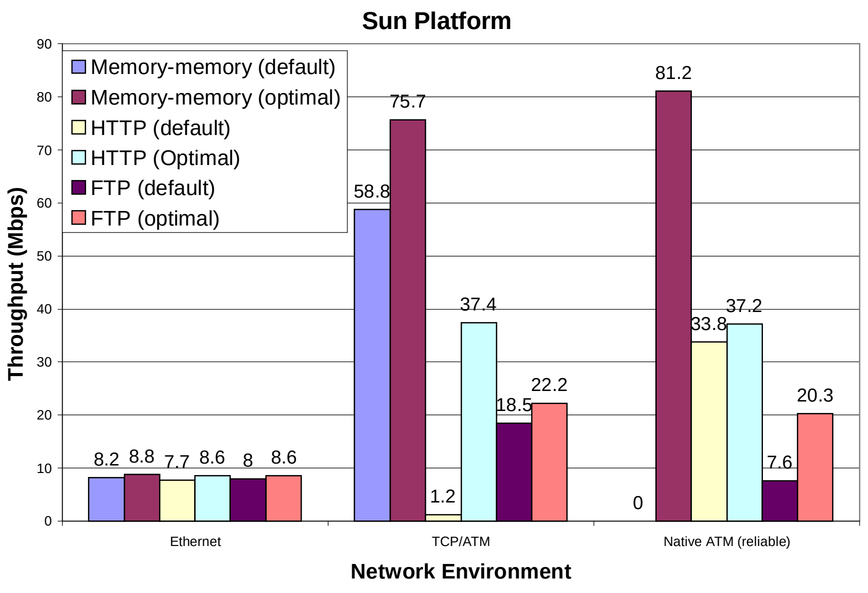


Figure 4.41: Summary of experimental results (Sun)

Chapter 5

Conclusions and Future Work

In this thesis, we evaluate, compare, and improve the throughput achieved by the HTTP and FTP applications when executing using three different network environments: TCP/IP over Ethernet, TCP over ATM (TCP/ATM) and native ATM using AAL5. We also use the Netperf and TTCP network benchmarks to study the characteristics of the three network environments with respect to throughput. In this chapter, our contributions are summarized and possibilities for future work are described.

5.1 Contributions

Modifications to existing network benchmarks and applications

One major contribution of this thesis is the porting of several network applications to the native ATM network environment. These applications include the TTCP and Webstone benchmarks, the NCSA HTTPd and Wu-ftpd daemons, and the FreeBSD FTP program.

Because the native ATM protocol does not ensure that data transfers are performed reliably, we have also implemented a simple window-based reliable flow-control mechanism within the native ATM applications to ensure reliable data transfers. This protocol is designed to be simple, and the experiments conducted on our local area network demonstrate that the protocol is capable of preventing data loss without significantly impeding performance.

Network benchmark results

Because the platforms we examined may be tuned for a particular network environment, the default network parameters provided by the system may not be suitable for other network environments. It is also very likely that the Fore TCP/ATM and native ATM networks were not taken into consideration when the default values were chosen.

We have found that poor choices of network or application parameters result in poor performance. For example, in our experiments which transfer data from the memory of an application on one host to the memory of application on another host (we call this "memory-to-memory transfer"), we have shown that by using the maximum possible socket buffer and ATM MTU sizes, combined with a 16384 byte (16 KB) data buffer, we are able to obtain a throughput of only 44.8 Mbps. Changing the data buffer size only slightly to 16240 bytes, with the use of the same socket buffer and ATM MTU sizes, a much higher throughput of 75.7 Mbps is obtained.

We have shown that by suitably adjusting the socket and data buffer sizes, as well as the ATM MTU size, significant improvements in throughput can be obtained. For example, by using the Ethernet and the Netperf benchmark with 8760 byte send and receiver socket buffers rather than the default 60 KB buffer, the SGI's peak throughput can be improved from 6.46 Mbps to 8.13 Mbps. Generally, we have found that maximizing the ATM MTU size should optimize the throughput. Maximizing the data buffer size can also optimize the throughput, but the use of system resources must be taken into consideration. Unfortunately, we have not been able to come up with a general rule for setting an optimal socket buffer size.

We have identified sets of optimal socket buffer, data buffer, and ATM MTU sizes that yield peak throughput in memory-to-memory transfers in each network environment. Using these optimal combinations, we show that, while throughput of the memory-to-memory transfer is limited by the Ethernet network bandwidth of 10 Mbps, throughput of up to 121 Mbps on the SGI's and 81 Mbps on the Sun's can be achieved using the TCP/ATM and native ATM environments. Although these throughputs are not close to the theoretical upper bound of 139 Mbps for our ATM environment, they are significantly higher than the throughput we were able to obtain using the HTTP and FTP applications.

From the experiments which measure the file access rates, we have found that, using uncached files, the maximum file read/write access rates of between 24 Mbps to 27 Mbps can be achieved by the file systems on the SGI's and Sun's. The file access rate decreases significantly when extremely small files are used (e.g., less than 1 Mbps with the use of the 500 byte file). This is due to the amount of overhead introduced by the file systems and disk overheads (e.g., system calls, seeks and rotational latency). This clearly suggests that, in real systems, the transfer of small uncached files will not be limited by network bandwidth but by the file system.

Network application results

Results from the FTP transfer experiments in which uncached files are used, show that the throughput obtained using the Ethernet is bound by the network bandwidth, whereas the throughput obtained using the TCP/ATM, unreliable and reliable native ATM environments is limited by the file access rate of the underlying file system and disks.

The HTTP transfer experiments, which are designed to measure throughput using different file sizes in the presence of file caching, shows that throughput decreases significantly as the file size decreases. It is reasonable to believe that this poor performance is not caused by the file systems, since the read access rates of all cached file sizes are much higher (it is likely that the files used in the HTTP experiments are automatically cached on the server). We believe that the inclusion of the HTTP request time in the overall transfer time is likely a major cause, since this time is proportionally large for small files. These results clearly indicate that, even with the presence of high-performance file systems and disks, poor HTTP performance can still result, in particular when extremely small files (500 byte and 5 KB files in our experiments) are transferred.

With the exception of the 5 KB and 50 KB files using the Sun's and the Ethernet environment, our other HTTP and FTP transfer experiments show that the throughput obtained using the optimal parameters is either equal to or better than the throughput obtained using the default parameters. For example, results from our HTTP experiments show that the average client throughput obtained using the reliable native ATM environments and a 5 MB file increases from 59.2 Mbps with the default parameters to 73.4 Mbps with the optimal parameters (an improvement of a factor 1.24). The use of the optimal data buffer size in the reliable native ATM environment is also shown to reduce the number of retransmissions. These results clearly show that, not only can the suitable adjustment of the software configurable parameters improve the throughput in memory-to-memory transfers (as obtained by the simple network benchmarks), but it can also improve and is critical to the performance of real network applications.

By comparing the throughput obtained from the memory-to-memory transfer and file access experiments with the throughput obtained by the two network applications, we observe that the former does not necessarily reflect the latter when applied to real network applications. This is clearly shown in the only case where the default parameters out-perform the optimal parameters when using the network applications. However, the performance of the simple benchmarks can be used to provide insight into the performance that might be possible for the network applications by setting an upper bound on performance. For example, we have found in our memory-to-memory transfer experiments that the peak throughputs using the Ethernet on the SGI's and Sun's are 8.1 Mbps and 8.8 Mbps, respectively. All throughputs in the corresponding HTTP and FTP experiments are observed not to exceed these upper bounds.

Platforms

The various throughput curves depicted from the results of our memory-to-memory transfer experiments clearly indicate that the SGI and Sun platforms exhibit very different behaviours with respect to throughput. Different throughputs are also obtained on the two platforms using the HTTP and FTP applications, as well as with the file access experiments in which only the operating systems and the file systems are involved. We believe that the difference in operating system designs, file systems, and hardware are some of the major causes of these different throughputs. We have not investigated this issue further since it is not an objective of our thesis.

Application Layer Protocols

The experiments for evaluating the performance of HTTP and FTP are not designed so that their results can be compared with each other. While the HTTP transfer experiments time both the HTTP requests and responses on potentially cached files, the FTP transfer experiments time only the FTP responses on uncached files. However, a few similarities between the results of the two sets of experiments can be seen. For example, we observe that in both cases, throughput decreases as the file size decreases, and that throughput using the Ethernet is bound by its limited bandwidth.

General Conclusions

Overall results show that running network applications such as the HTTP and FTP servers and clients using a reliable native ATM environment as opposed to a conventional local area network such as Ethernet, does provide significantly increased network throughput at the application level. For example, our HTTP experiments show that on the SGI platform, throughput improves by about ten times when using the reliable native ATM environment in comparison with the Ethernet environment.

Results from our HTTP and FTP experiments, however, show that the throughput achieved by these applications when using the TCP/ATM and native ATM environments is still far below that which the physical network medium can provide. In order for these applications to benefit from the full potential of the ATM and other high-speed networks, further improvements in other hardware and software components of the platform such as the file systems and disks must be performed.

Results from the HTTP experiments show that better throughput is obtained on the SGI's when using the native ATM environment with our reliable native ATM protocol (e.g., 73.4 Mbps when using the 5 MB file) as opposed to the TCP/ATM environment (e.g., 48.2 Mbps when using the 5 MB file). On the other hand, the corresponding results on the Sun's show that throughputs obtained when using both environments are similar (about 37 Mbps). Furthermore, considering that in the FTP experiments, throughputs obtained when using the two environments are greatly restricted by the file systems, we believe that the efforts involved in writing native ATM applications are not practical at this time, especially considering that TCP has been around for more than a decade and a large number of applications already use it.

Meanwhile, we believe that the combination of well constructed TCP/ATM applications and well tuned network and application parameters can be used to obtain considerable improvements in throughput (especially when compared with the Ethernet environment). However, note that average disk drive, operating system, and file system construction currently impose limits to the expected bandwidths (no higher than 27 Mbps on the SGI and Sun platforms we have examined). As a result, we believe that significant improvements are required in these areas if these network-based applications are to achieve significantly higher throughput.

All our experiments are done in uncongested environments. Different conclusions may be drawn if network congestion is taken into consideration. In particular, we expect the efficiency in handling congestion to be different between TCP/ATM and native ATM network environments.

5.2 Future Work

In this thesis, we have concentrated on analyzing the performance of network applications when executing in a local area network environment. A natural extension of this work would be to evaluate these applications in a wide area network environment. We expect that our results may change in such an environment because of its different characteristics (e.g., higher latency and higher error rates).

Due to the intractability of testing all combinations of network parameters, we have chosen those which we believe to provide a good starting point. Future studies involving the tuning of other other network parameters such as TCP MSS for Ethernet and TCP/ATM, and QoS for native ATM may also prove to be interesting.

We have concentrated our work on the two most frequently used network protocols (HTTP and FTP). Studies on other commonly used network protocols such as NFS and SMTP are a possible area of future work. In addition, it would be interesting to consider platforms other than the SGI's and Sun's, for example, the popular Intel-based PC's using Microsoft software.

Because the applications we examined are mostly concerned with obtaining high throughput, our work has concentrated on throughput measurements. Further investigation of other performance metrics such as latency should provide insight that would be valuable to applications whose performance depends on such latencies.

Our work has considered only a single client and single server executing simultaneously. An interesting and natural progression of our work would be to examine the impact of network congestion. This will certainly provide insights into a popular claim that the switch-based ATM network architecture can handle network congestion much better than conventional shared-medium network architecture.

Results from the HTTP transfer experiments show that poor throughput is achieved when using small files, with an exact cause unknown to us (we believe that the HTTP request time is very likely a major factor in this poor performance). At the time of writing, HTTP/1.1 [10] is in its last stage of development. In order to reduce protocol overhead and network latency, it is designed to extend the life-span of the TCP connections in which the HTTP requests and responses are made. It would be interesting to study how the performance of applications is impacted by HTTP/1.1.

In this thesis, we have examined several issues related to the performance of network applications when executing over Ethernet and ATM networks. While there are many possible avenues for future work, we believe that we have provided a good starting point into this area of research.

Bibliography

[1] I. Andrikopoulos, T. Örs, M. Matijaševiċ, H. Leitold, S. Jones, and R. Posch, "TCP/IP Throughput Performance Evaluation for ATM Local Area Networks," in *Proceedings of the 4th IFIP Workshop on Performance Modelling and Evaluation of ATM Networks*, Ilkley, UK, pp. 72/1–72/15, July 1996.

[2] Anixter Incorporated, "Ethernet Switching," White Paper, 1997. Currently available at `http://www.anixter.com/techlib/whiteppr/network/anixeswp.htm`.

[3] Apache Group, *Apache HTTP Server Version 1.3*, May 1998.
Currently available at `http://www.apache.org/docs/`.

[4] G. Armitage and K. Adams, "How Inefficient is IP over ATM anyway?," *IEEE Network*, Vol. 9, No. 1, pp. 18–27, January 1995.

[5] R. Atkinson, "Default IP MTU for Use over ATM AAL5," *RFC1626*, May 1994.

[6] ATM Forum Technical Committee, "LAN Emulation over ATM Version 1.0," *Specification af-lane-0021.000*, January 1995. Currently available at
`ftp://ftp.atmforum.com/pub/approved-specs/af-lane-0021.000.ps`.

[7] ATM Forum Technical Committee, "Multi-Protocol Over ATM Specification V1.0," *Specification af-mpoa-0087.000*, July 1997. Currently available at
`ftp://ftp.atmforum.com/pub/approved-specs/af-mpoa-0087.000.ps`.

[8] N. Aucoin, "Evaluating the Performance of HTTP over ATM versus Ethernet," Course Project, Parallel and Distributed Computing, York University, Ontario, May 1996.

[9] T. Berners-Lee, R. Fielding, and H. Frystyk, "HyperText Transfer Protocol — HTTP/1.0," *RFC1945*, May 1996.

[10] T. Berners-Lee, R. Fielding, H. Frystyk, J. Gettys, and J. Mogul, "HyperText Transfer Protocol — HTTP/1.1," *RFC2068*, January 1997.

[11] R. Braden, L. Zhang, S. Berson, S. Herzog and S. Jamin, "Resource ReSerVation Protocol (RSVP) — Version 1, Functional Specification," *RFC2205*, September 1997.

[12] A. Carlton, "An Explanation of the SPECweb96 Benchmark," White Paper, Standard Performance Evaluation Corporation, December 1996. Currently
available at `http://www.specbench.org/osg/web96/webpaper.html`.

[13] J. Cavanaugh, "Protocol Overhead in IP/ATM Networks," Technical Report, Minnesota Supercomputer Center, Inc., Minneapolis, August 1994.

[14] S. Chang, D. Du, J. Hsieh, M. Lin, and R. Tsang, "Enhanced PVM Communications over a High-Speed Local Area Network," *IEEE Parallel and Distributed Technology*, Vol. 3, No. 3, pp. 20–32, Fall 1995.

[15] D. Clark, "Window and Acknowledgment Strategy in TCP," *RFC813*, July 1982.

[16] D. Clark, V. Jaconbson, J. Romkey, and H. Salwen, "An Analysis of TCP Processing Overhead," *IEEE Communications Magazine*, No. 6, pp. 23–29, June 1989.

[17] D. Comer and J. Lin, "TCP Buffering and Performance over an ATM Network," *Internetworking: Research and Experience*, Vol. 6, No. 1, pp. 1–13, March 1995.

[18] J. Day and H. Zimmerman, "The OSI Reference Model," in *Proceedings of the IEEE*, pp. 1334–1340, December 1983.

[19] A. Dick, "FTP Executing over Ethernet and ATM Networks," Undergraduate Thesis, Space and Communication Sciences, York University, Ontario, April 1996.

[20] Digital Equipment Corporation, Intel Corporation, and Xerox Corporation, "The Ethernet, Physical and Data Link Layer Specifications," *Version 2.0*, 1982.

[21] I. N. Division, "Netperf: A Network Performance Benchmark, Revision 2.1," Technical Report, Hewlett-Packard, February 1996. Currently available at `ftp://ftp.cup.hp.com/dist/networking/benchmarks/netperf/netperf-2.1pl3.tar.gz`.

[22] Fore Systems Incorporated, *ForeRunner SBA-100/-200 ATM SBus Adapter — User's Manual*, October 1994.

[23] Fore Systems Incorporated, *Fore IP — ForeThought 3.0.1 (BETA Release)*, July 1995.

[24] Fore Systems Incorporated, *ForeRunner ASX-200BX/ASX-200BXE ATM Switch — User's Manual*, May 1995.

[25] Fore Systems Incorporated, *ForeRunner GIA-100/-200 ATM GIO Bus Adapter for Silicon Graphics — User's Manual*, June 1995.

[26] Y. Fouquet, R. Schneeman, D. Cypher, and A. Mink, "ATM Performance Measurement: Throughput Bottlenecks and Technology Barriers," in *Proceedings of the Gigabit Networking Workshop GBN '95*, April 1995.

[27] K. Frazer, "NSFNET: A Partnership for High-Speed Networking," Final Report 1987-1995, Merit Network, Inc., 1995.
Currently available at `http://www.merit.edu/nsfnet/final.report/`.

[28] B. Goodheart and J. Cox, *The Magic Garden Explained: The Internals of UNIX System V, Release 4*. Prentice Hall, 1994.

[29] J. Graham, *Solaris 2.x: Internals and Architecture*. McGraw-Hill, 1995.

[30] J. Heinanen, "Multiprotocol Encapsulation over ATM Adaptation Layer 5," *RFC1483*, July 1993.

[31] C. Huang and P. McKinley, "Communication Issues in Parallel Computing across ATM Networks," *IEEE Parallel and Distributed Technology*, Vol. 2, No. 4, pp. 73–86, Winter 1994.

[32] Institute of Electrical and Electronics Engineers, "Carrier Sense Multiple Access with Collision Detection," *IEEE Standard 802.3*, 1985.

[33] ITU-T, "B-ISDN Protocol Reference Model and its Application," *Recommendation I.321*, April 1991.

[34] M. Kawarasaki and B. Jabbari, "B-ISDN Architecture and Protocol," *IEEE Journal on Selected Areas in Communications*, Vol. 9, No. 9, pp. 1405–1415, December 1991.

[35] C. Kent and J. Mogul, "Fragmentation Considered Harmful," in *Proceedings of the ACM SIGCOMM '87 Workshop on Frontiers in Computer Communications Technology*, pp. 390–401, August 1987.

[36] T. Kwan, R. McGrath, and D. Reed, "NCSA's World Wide Web Server: Design and Performance," *IEEE Computer*, Vol. 28, No. 11, pp. 69–74, November 1995.

[37] O. Kyas, *ATM networks.* International Thomson Computer Press, 1995.

[38] M. Laubach, "Classical IP and ARP over ATM," *RFC1577*, January 1994.

[39] G. Lehey, *The Complete Freebsd.* Walnut Creek, September 1996.

[40] T. Luckenbach, R. Ruppelt, and F. Schulz, "Performance Experiments within Local ATM Networks," in *Proceedings of the 12th European Fiber Optic Communication and Networks Conference*, Heidelberg, Germany, June 1994.

[41] M. Matijaševič, R. Posch, and F. Pucher, "ATM LAN Throughput Measurements and Evaluation for BSD socket and FSI Application Programming Interface," in *Third Workshop on Performance Modelling and Evaluation of ATM Networks — Participants Proceedings (Technical Programme)*, Ilkey, West Yorkshire, UK, pp. 64/1–64/10, July 1995.

[42] R. McGrath, "Performance of Several HTTP Demons on an HP 735 Workstation," Technical Report, Computing and Communications, NCSA, April 1995.

[43] R. McGrath, "Performance of Several Web Server Platforms," Technical Report, Computing and Communications, NCSA, January 1996.

[44] M. McKusick, K. Bostic, M. Karels, and J. Quarterman, *The Design and Implementation of the 4.4BSD Operating System.* Addison Wesley, 1996.

[45] K. Moldeklev and P. Gunningberg, "How a Large ATM MTU Causes Deadlocks in TCP Data Transfers," *IEEE/ACM Transactions on Networking*, Vol. 3, No. 4, pp. 409–422, August 1995.

[46] P. Newman, T. Lyon, and G. Minshall, "Flow Labelled IP: A Connectionless Approach to ATM," in *Proceedings of IEEE Infocom '96*, pp. 1251–1260, March 1996.

[47] C. Partidge, *Gigabit Networking.* Addison-Wesley, 1994.

[48] J. Postel, "User Datagram Protocol," *STD6, RFC768*, August 1980.

[49] J. Postel, "Internet Protocol," *STD5, RFC791*, September 1981.

[50] J. Postel, "Transmission Control Protocol," *STD7, RFC793*, September 1981.

[51] J. Postel and J. Reynolds, "Telnet Protocol Specification," *STD8, RFC854*, May 1983.

[52] J. Postel and J. Reynolds, "File Transfer Protocol (FTP)," *STD9, RFC959*, October 1985.

[53] H. Schulzrinne, S. Casner, R. Frederick, and V. Jacobson, "RTP: A Transport Protocol for Real-time Applications," *RFC1889*, January 1996.

[54] Silicon Graphics, Incorporated, *IRIX 6.2 Release Notes*, 1995.

[55] Silicon Graphics, Incorporated., *README WebStone 2.0.1 Bug Fix Release*, June 1996.

[56] M. St. Johns, "Authentication Server," *RFC931*, January 1985.

[57] W. R. Stevens, *TCP/IP Illustrated*, Vol. 1. Addison Wesley, May 1994.

[58] W. R. Stevens, *UNIX Network Programming*, Vol. 1. Prentice Hall, 1998.

[59] W. R. Stevens and G. Wright, *TCP/IP Illustrated*, Vol. 2. Addison Wesley, January 1995.

[60] G. Thompson, "Standards: Work Progresses on Gigabit Ethernet," *IEEE Computer*, Vol. 30, No. 5, pp. 95–96, May 1997.

[61] K. Thompson, M. Gregory, and R. Wilder, "Wide-Area Internet Traffic Patterns and Characteristics," *IEEE Network*, Vol. 11, No. 6, pp. 10–23, November 1997.

[62] G. Trent and M. Sake, "Webstone: The First Generation in HTTP Server Benchmarking," White Paper, Silicon Graphics, Inc., 1995.
Currently available at `http://www.mindcraft.com/webstone/paper.html`.

[63] UNIX International, "Data Link Provider Interface Specification," *Revision 2.0.0*, August 1991.

[64] US Army Ballistic Research Lab, "TTCP - Test TCP and UDP Performance," White Paper, December 1984.
Currently available at `ftp://ftp.arl.mil/pub/ttcp/`.

[65] I. Vessey and G. Skinner, "Implementing Berkeley Sockets in System V Release 4," in *Proceedings of the Winter 1990 USENIX Technical Conference*, pp. 177–193, January 1990.

[66] Washington University, "Washington University FTP Server," White Paper, September 1997.
Currently available at `ftp://ftp.academ.com/pub/wu-ftpd/`.

[67] A. Wolman, G. Voelker, and C. Thekkath, "Latency Analysis of TCP on an ATM Network," in *Proceedings of the Winter USENIX Conference*, San Francisco, CA, pp. 167–179, January 1994.