Identifying and Analyzing Sources of Overhead in the TCP/IP Communication Protocol Over a Local Area Network

by

Qizhong Chen

Thesis submitted to the Faculty of the Virginia Polytechnic Institute and State University in partial fulfillment of the requirements for the degree of Master of Science in Department of Computer Science

APPROVED:

Marc Abrams, Ph.D., Chairman

Scott F. Midkiff, Ph.D. Dennis G. Kafura, Ph.D.

July, 1991

Blacksburg, Virginia
Identifying and Analyzing Sources of Overhead in the TCP/IP Communication Protocol Over a Local Area Network

by

Qizhong Chen

Marc Abrams, Ph.D., Chairman
Department of Computer Science

(ABSTRACT)

This research investigates various aspects of user-perceived network performance over a local area network for two transport layer protocols: TCP and UDP. The sensitivity of user-level performance to the choice of different speed hosts, host loads, and application program interfaces are studied. Our measurements serve as a guide in designing performance critical applications. Moreover, we present a detailed timing analysis of the dynamic behavior of the TCP/IP implementation in the MD-DOS/IP package. The analysis shows that the TCP flow control mechanism has a severely negative impact on the performance. The TCP data copy and checksum are the major overheads of TCP segment processing. Finally, the bottleneck of data communication using TCP/IP is identified based on queueing theory and empirical measurements.
Acknowledgements

I would like to express my sincere appreciation to my major professor, Dr. M. Abrams, for his willing assistance and guidance throughout my program of study. I am also grateful for his friendship and many stimulating conversations we have had together discussing problems in computer science.

A special thanks to Dr. S. Midkiff and Dr. D. Kafura for their helpful suggestions and for serving on my Advisory Committee.

Many thanks to Jay Wang and Xiaodong Zhang for their encouragement during the ups and downs of this study.

I am grateful for the financial support received from the Zheng Ge Ru Foundation which enabled me to pursue this degree.

Appreciation also is due to the Computer Science Department at Radford University for providing computer facilities on which some experiments of this research were carried out.

Finally, I am most grateful to my parents and my brother for their continued encouragement.
## Table of Contents

### Chapter 1. Introduction

1.1 Network Protocols .................................................. 2
   1.1.1 The OSI Reference Model ..................................... 2
   1.1.2 TCP, UDP and IP Protocols .................................. 5
1.2 Motivation for this Research ....................................... 8
1.3 Related Work .................................................................. 10
1.4 Summary ........................................................................ 13

### Chapter 2. Experimental Environment

2.1 Hardware Components for the Research .......................... 14
2.2 Interprocess Communication Software ......................... 15
   2.2.1 Sockets ............................................................ 15
   2.2.2 TLI ................................................................. 19
2.3 Writing Size ............................................................... 22
2.4 Repetition Count ......................................................... 23
2.5 Artificial Host Load ...................................................... 24
2.6 MD-DOS/IP Package .................................................... 24
2.7 PC Measurement Card ................................................. 25
2.8 Summary ................................................................. 25

Chapter 3. Performance Tests ................................................. 26
3.1 Introduction ............................................................... 26
  3.1.1 Overview of the Tests .............................................. 26
  3.1.2 Performance Metrics .............................................. 28
3.2 Sensitivity of User-level Performance to Choice of Host .......... 29
  3.2.1 Test Procedure .................................................... 29
  3.2.2 Test Results and Observations .................................. 29
3.3 Sensitivity of User-level Performance to Host Load ............. 37
  3.3.1 Test Procedure .................................................... 37
  3.3.2 Test Results and Observations .................................. 40
3.4 Sensitivity of User-level Performance to Choice of Application Program Interface .... 45
  3.4.1 Test Procedure .................................................... 45
  3.4.2 Tests Results and Observations .................................. 45
3.5 Summary ................................................................. 48

Chapter 4. TCP/IP Overhead Study ......................................... 49
4.1 Instrument the TCP/IP Implementation ............................... 49
  4.1.1 The Internal Structure of the TCP/IP Implementation .......... 49
  4.1.2 Modifications to TCP, IP and Driver Routines .................. 53
4.2 The Overhead of TCP/IP Packet Processing ......................... 57
  4.2.1 Test Procedure .................................................... 57
  4.2.2 Test Results and Observations .................................. 58
  4.2.3 Some Speed Predictions ........................................... 63
4.3 The Overhead of TCP Segment Processing .......................... 64
  4.3.1 Test Procedure .................................................... 65

Table of Contents
### List of Illustrations

<table>
<thead>
<tr>
<th>Figure</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>OSI 7-layer model</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Layering in the TCP/IP protocol suite</td>
<td>6</td>
</tr>
<tr>
<td>3</td>
<td>Socket system calls for connection-oriented communication</td>
<td>18</td>
</tr>
<tr>
<td>4</td>
<td>Socket system calls for connectionless communication</td>
<td>20</td>
</tr>
<tr>
<td>5</td>
<td>TCP throughput on different machine pairs</td>
<td>31</td>
</tr>
<tr>
<td>6</td>
<td>UDP throughput on different machine pairs</td>
<td>35</td>
</tr>
<tr>
<td>7</td>
<td>Comparison of throughput for TCP and UDP (on MacII and A3000)</td>
<td>38</td>
</tr>
<tr>
<td>8</td>
<td>Comparison of throughput for TCP and UDP (on NeXT, SUN, and DECstation)</td>
<td>39</td>
</tr>
<tr>
<td>9</td>
<td>Comparison of throughput for sockets and TLI on A3000</td>
<td>47</td>
</tr>
<tr>
<td>10</td>
<td>Internal structure of the TCP/IP implementation</td>
<td>50</td>
</tr>
<tr>
<td>11</td>
<td>TCP/IP calling hierarchy for sending a packet</td>
<td>52</td>
</tr>
<tr>
<td>12</td>
<td>TCP/IP calling hierarchy for receiving a packet</td>
<td>54</td>
</tr>
<tr>
<td>13</td>
<td>The behavior of TCP/IP on the sending side (D310 to PS2)</td>
<td>59</td>
</tr>
<tr>
<td>14</td>
<td>Overhead of TCP segment processing on the sending side</td>
<td>67</td>
</tr>
<tr>
<td>15</td>
<td>Overhead of TCP segment processing on the receiving side</td>
<td>70</td>
</tr>
<tr>
<td>16</td>
<td>The receiving buffer size as a function of &amp;rho.</td>
<td>73</td>
</tr>
<tr>
<td>17</td>
<td>The TCP throughput for different machine pairs (Dell310 as the receiver)</td>
<td>77</td>
</tr>
<tr>
<td>18</td>
<td>The TCP throughput for different machine pairs (DEC2100 as the receiver)</td>
<td>79</td>
</tr>
</tbody>
</table>
List of Tables

Table  1. Machine configurations used for experiments .................. 16
Table  2. Comparison of sockets and TLI system call ..................... 21
Table  3. The data transfer delay to send 40 Kbytes on TCP ............... 30
Table  4. The data transfer delay to send 40 Kbytes on UDP ............... 34
Table  5. Data transfer delay for varying host load (TCP) ............... 41
Table  6. Data transfer delay for varying host load (UDP) ............... 42
Table  7. Data transfer delay for 512,000 bytes on MacII .................. 44
Table  8. Data transfer delay for Berkeley sockets and System V TLI on A3000 .... 46
Table  9. The system idle time on the sending side for total time to send 40 Kbyte data ... 62
Table 10. Cost for execution of TCP/IP protocol for a 1024 byte segment .......... 65
Chapter 1. Introduction

Computer networks have revolutionized the use of computers. Some benefits of computer networking are [1]:

- resource sharing by making all programs, data and equipment available to anyone on the network without regard to the physical location of the resource and the user,

- high reliability by having alternative sources of supply, and

- providing a powerful communication medium among widely separated people.


This research studies the user-level network performance when using TCP/IP protocol suite in an Ethernet-based environment. The effects of the following factors have on communication performance are examined in detail:

1. type of processor,
2. host load, and

3. the application program interface.

Also studied is the overhead of TCP/IP packet processing. Furthermore, the bottleneck of data communication using TCP/IP is studied, which yields insight into the issue of whether window or rate-based flow control is superior.

1.1 Network Protocols

1.1.1 The OSI Reference Model

Modern computer networks are designed and implemented in a highly structured way. Network software is organized as a series of layers, each one using the services provided by the lower layer and offering its own services to the upper layer. The purposes of the layer structure are that:

- each layer can be designed, implemented, and tested separately, hence reducing the total complexity of development and maintenance, and

- each layer can be changed or replaced without affecting the other layers if it retains the same interfaces.

The International Standard Organization (ISO) developed the Open System Interconnection (OSI) Reference Model [1] as a first step toward international standardization of the various protocols. The OSI model provides a detailed standard for describing a network. Figure 1 shows the OSI model.
Figure 1. OSI 7-layer model
The dashed lines in Figure 1 represent the protocols, which define the rules and conventions used in the conversation between peer layers on different machines [1].

The physical layer deals with transmitting raw bits over a communication channel. It defines the mechanical, electrical, and procedural interfaces [1].

The data link layer provides error-free transmission to the network layer based on the raw transmission facility. It converts the network data into data frames, and transmits the frames sequentially. Transmission errors are detected by having the receiver send back an acknowledgement to the sender. The layer also performs flow control, which ensures that a fast sender does not overwhelm a slow receiver [1].

The network layer is concerned with controlling the operation of the network. The main function of this layer is packet routing, which determines how packets are routed to the destination. When too many packets are present in the network, the network becomes incapable of delivering the packets and begins to lose packets. The result is a dramatic degradation of network performance. This phenomenon is called network congestion. Another function of the network layer is congestion control. Accounting and internetworking are two other functions [1].

The transport layer is an end-to-end layer, which means this layer concerns the conversation between the source machine and the destination machine. In the layers below the transport layer, the protocol is between each machine and its immediate neighbor. The transport layer provides error-free connection-oriented service or unreliable datagram service to transport users on the hosts. End-to-end flow control is also a responsibility of this layer. Other important functions include creating multiple network connections to achieve high throughput and multiplexing several transport connections onto one network connection to reduce the cost [1].

The session layer allows users to establish sessions. One of the services of the session layer is dialogue control which allows traffic to go in both directions at the same time, or in one direction.
at a time. Another session service is synchronization. Checkpoints are inserted into the data stream, so that after a crash, only the data after the last checkpoint needs to be retransmitted [1].

The presentation layer is concerned with the syntax and the semantics of the information transmitted. It provides data encoding, data compression, and cryptography services [1].

The application layer contains a variety of protocols that are commonly needed. By using the services provided by the lower layer, the application layer protocols provide various services to the user. OSI FTAM (File Transfer, Access and Management), MOTIS (Message-Oriented Text Interchange System) and JTM (Job Transfer and Management) are examples of application layer protocols [1].

1.1.2 TCP, UDP and IP Protocols

The transport layer is an important layer in the network protocol hierarchy. Its task is to provide reliable connection-oriented or unreliable datagram data transport services, independent of the quality of services provided by the underlying subnet. The Department of Defense Standard Transmission Control Protocol (TCP) [1] is a general-purpose transport layer protocol designed to provide reliable data transmission on an unreliable subnet. Another transport layer protocol, User Datagram Protocol (UDP) [4], provides unreliable connectionless data delivery service. Associated with TCP and UDP is a network layer protocol called Internet Protocol (IP) [3]. TCP (UDP)/IP is used in the Internet, including MILNET and NFSnet, and is the most popular transport protocol in U.S. commercial products. Figure 2 [5] shows the relationship of the protocols in the TCP (UDP)/IP protocol suite.

TCP provides reliable connection-oriented service. TCP is able to transfer a continuous stream of octets (bytes) in each direction between its users by packing some number of octets into segments for transmission through the network. Assuming the underlying network is unreliable, TCP detects
Figure 2. Layering in the TCP/IP protocol suite
and recovers from data loss, duplication, and corruption to provide reliable data transmission between end users. This is achieved by assigning a sequence number to each octet transmitted, and requiring a positive acknowledgement (ACK) from the receiving TCP. If the ACK is not received within a timeout interval, the data is retransmitted. At the receiver, sequence numbers are used to order segments that may be received out of sequence and to eliminate duplicates. Bit errors are handled by adding a checksum to each segment transmitted, checking it at the receiver, and discarding damaged segments. Another function of TCP is end-to-end flow control between the source and the ultimate destination. This function is essential because machines of various speeds and sizes communicate through networks and it is easily for a source machine to transmit data faster than the destination can receive the data. TCP provides a means for the receiver to govern the amount of data sent by the sender. This is achieved by returning an advised window with every ACK indicating a range of acceptable sequence numbers beyond the last segment successfully received. The window indicates an allowed number of octets that the sender may transmit before receiving further permission. Usually, the advised window specifies the receiver’s current buffer size available for this connection. The reliability and flow control mechanism described above require that TCP initialize and maintain certain status information for each data stream. The combination of this information, including sequence number and window size, is called a connection. When two processes wish to communicate, their TCP’s must first establish a connection. When their communication is complete, the connection is terminated or closed to free resources for other users [2].

UDP is another transport layer protocol. It provides unreliable connectionless data delivery service. Unlike TCP, it does not use acknowledgements to make sure data is correctly delivered nor does it use the window mechanism to control the flow. Thus UDP datagrams can be lost, duplicated, corrupted or arrive out of order. UDP’s main function is to deliver the datagram to the correct recipient among multiple destinations within a given host computer. Most computers support multitasking, which means multiple processes could execute simultaneously. It is essential to provide a way to specify a particular process on a particular machine to be the ultimate destination. This is achieved by binding a unique protocol port to each inter-process communication. Data re-
ceived from a protocol port is given to the process associated with the port. UDP provides protocol ports used to distinguish among multiple processes executing on a single machine. To summarize, UDP allows multiple application programs executing on a given host to send and receive data independently.

IP provides an unreliable connectionless datagram service. It has no acknowledgement, retransmission and flow control. There is no error control for data, only a header checksum. Therefore, IP does not guarantee the correct delivery of data because a datagram may be lost, be corrupted, or exceed its lifetime and be discarded. Errors detected may be reported via the Internet Control Message Protocol (ICMP). IP's main function is routing and packet fragmentation. One of IP's purposes is to move datagrams through an interconnected set of networks. Datagrams are routed from one internet node to another based on the interpretation of the internet address. One of IP's tasks is to map the internet address to local net address. Another function of IP is to break datagrams into smaller units called fragments when passing data on a network with a smaller maximum transmission unit. IP is also responsible for reassembling the fragments into the original datagram at the destination.

1.2 Motivation for this Research

Computer networks facilitate very powerful computing environments for users. At the same time networks present new challenges for a user because there are many factors that affect communication performance. It is obvious that different processors, varying host loads, and different user/protocol interfaces affect the communication across machines. An assessment of the impact these factors have on the performance is needed if we want to efficiently use the inter-process communication mechanism provided in a network environment. This research investigates the
performance of TCP and UDP protocols for different hosts, varying host load, and varying user/protocol interfaces with emphasis on throughput and delay.

In the past, when the communication media worked at speeds between 110 and 9600 bps (bits per second), the communication channel itself was often the bottleneck. Today, high speed local area networks, especially fiber optical networks, have moved the bottleneck from communication channels to the rate at which hosts and gateways can send and receive data [6]. When using high speed local area networks, users often perceive a throughput well below that offered by the network. Since the TCP/IP suite has considerable functionality and is typically executed in software by a host machine, there is concern whether a TCP/IP implementation can meet performance requirements of application programs. It is easy to blame the TCP/IP protocol for disappointing performance. But to fully understand the performance problems of TCP/IP, we need to determine how much overhead is due to the protocol, how much is due to a particular implementation, and how much is due to other factors. Only after identifying the sources of the TCP/IP overhead, can we improve its performance. This research studies a TCP/IP implementation, the MD-DOS/IP package, in detail. The MD-DOS/IP package was developed at the University of Maryland for IBM PC and PS/2 computers. The TCP/IP implementation is instrumented and the time each packet spends on different layers is recorded. By analyzing these time traces, we identify the overhead of the TCP/IP implementation. Based on our study, performance predictions of TCP/IP running on other machines and implementations suggestions are made.
1.3 Related Work

Network performance studies are usually carried out for two reasons:

- to understand a particular system well so as to achieve optimal performance, and

- to identify the performance bottlenecks so as to improve overall network performance.

Several works reporting the performance of the TCP(UDP)/IP protocol are discussed below.

Based on the observations of distributed graphics applications running on the V distributed operating system, Lantz [7] shows the effect of the following parameters (listed in order of their importance) on the performance of data transmission.

- Speed of the source machine.

- Speed of the destination machine.

- Choice and implementation of network transport protocol (e.g., choose a general-purpose transport protocol versus a specialized protocol; implement the transport process within the operating system kernel versus outside the kernel).

- Pattern of data transmission (i.e. the granularity of each data transfer).

Svobodova [8] reports the measured throughput and delay of several transport layer protocols. These measurements indicate that performance improvements (throughput and delay) of various OSI TP4/CLNS and TCP/IP implementations can be achieved by implementation optimizations. A major TCP/IP performance improvement is achieved by a combination of many changes:
\begin{itemize}
\item elimination of data copies into internal TCP buffers,
\item modification of network drivers to allow more than one packet to be queued for transmission,
\item prediction of the header of the next packet to be sent or received, and
\item caching of frequently used data structures.
\end{itemize}

The author concludes that good design of communication software, clever coding of the protocol, and the support of the operating system play a critical role in improving the transport service.

Cabrera, et al. [9] study user-perceived performance when using TCP/UDP on 4.2 BSD UNIX in an Ethernet environment. They assess the impact that different processors, network hardware interfaces, Ethernets, and host load have on the performance. The host load and network hardware interface have a severe effect on the user processes' perception of network throughput. They also analyze the 4.2 BSD UNIX TCP/UDP implementation and find that data copying and checksumming account for a disproportionate share of the total delay time.

In this research, we also assess the impact that different processors and host load have on the performance. Moreover, we measure the effect that varying application program interfaces have on the performance.

The following works discuss the TCP(UDP)/IP protocol and its implementation.

Clark, et al. [10] discuss problems with TCP flow control strategy, which is a window mechanism, and conclude that there are two major problems with the window mechanism. First, the flow control in the window mechanism is vulnerable to transmission errors and delays because the window mechanism combines both data flow control and error recovery. When the transmission error rate and the network delay is high, the sender must stop frequently to wait for the acknowledgment. Under such a situation, the window mechanism no longer controls the flow. Second, the window
mechanism uses the receiver advised window size as the control parameter. But the advised window size does not carry enough information for flow control. This is because the window controls how much data can be sent rather than how fast the transmission should go. Transmissions occurring at an unregulated rate can easily congest the network and the receiver. In this case the window mechanism negatively influences performance.

Sanghi, et al.[11,12,13] describe an instrumentation of 4.3BSD UNIX TCP/IP. They conclude that a high resolution clock is essential to obtain good round trip time estimates, and that the round trip time estimator suggested by Jacobson [14] performs better than the one suggested in the original TCP specification [2].

Clark [15] describes implementation strategies to avoid the so-called silly window syndrome (SWS) and excess acknowledgements.

Tips on efficiently implementing TCP/IP on personal computers are given by Saltzer [16]. A new interface technique, the upcall, is proposed to avoid the excessive interface code that results from layering. Enhancements of DOS necessary to support network applications are also discussed.

Clark, et al. [17] investigate whether the transport layer is the bottleneck of data transmission. The technique used is to identify the normal path through the 4.3BSD UNIX TCP and count the instructions. Based on their study, they predict that TCP can be used on high speed fiber optical networks if implemented properly. The major overhead of TCP/IP packet processing comes from the operating system interface, which handles interrupts, restarts I/O devices, wakes up processes, and sets timers. Another overhead is operations that touch bytes, such as memory copy and checksum.

Our work is similar to Clark, et al. [17] in its objective of identifying the major contributions to software processing of network code. But we differ because we actually use a high resolution timer.
to measure the cost associated with the TCP/IP and we examine idle periods that are forced by the TCP window mechanism.

1.4 Summary

This chapter discussed the OSI network model and the Department of Defense's TCP, UDP and IP protocols. The motivations for this research were also presented. This was followed by a brief survey of related work.
Chapter 2. Experimental Environment

The test environment for this research is described in this chapter. The tests are carried out on various workstations and personal computers whose characteristics are presented in this chapter. The chapter discusses the program designed to determine the performance properties of user process communication based on TCP and UDP. To identify the overhead of TCP/IP packet processing, the TCP/IP implementation of the MD-DOS/IP package is instrumented. Features of the MD-DOS/IP package are presented. This is followed by a description of the PC measurement card [18] used in the tests.

2.1 Hardware Components for the Research

The machines used for the research are IBM-PS/2 model 80, IBM-compatible Dell/310, Commodore Amiga 3000UX, Apple Macintosh II, NeXTcube, NeXTstation, SUNstation 4/390, DECstation 2100, and DECstation 5000/200. These machines are connected via an IEEE 802.3
Carrier Sense Multiple Access with Collision Detection (CSMA/CD) 10 megabit/second local area network. The selected machines are listed in Table 1. The table shows the CPU speed and operating system on the machines. These configuration characteristics are crucial for interpreting the performance results in subsequent chapters.

2.2 Interprocess Communication Software

The performance testing software is based on two programs: a data sending program and a data receiving program. The sending program transmits data as fast as possible without waiting for the acknowledgement of the previous data. It is used to determine how fast the network software can be driven. The receiving program serves as a data sink; it receives data from the transport layer and discards it. The receiving program also records how much data is received and the time spent in doing so (the time that elapses from the instant the receiving program receives the first user message until the instant the recipient receives the last user message).

Berkeley UNIX sockets and AT&T UNIX System V Transport Layer Interface (TLI) are two commonly used application program interfaces to the communication protocol [5]. These two interfaces are used in the testing program. There are four versions of the performance testing program: TCP with sockets, UDP with sockets, TCP with TLI, and UDP with TLI. The source code for each version is included in Appendices A through D.

2.2.1 Sockets

A communication link between two processes is specified by a 5-tuple [5]:

\[ \{ \text{protocol, local-addr, local process, foreign-addr, foreign process} \} \]
<table>
<thead>
<tr>
<th>Host Name</th>
<th>Machine Type</th>
<th>CPU</th>
<th>Operating System</th>
<th>TCP receiving buffer size</th>
</tr>
</thead>
<tbody>
<tr>
<td>D310</td>
<td>Dell/310</td>
<td>Intel-80386 12.5MHz</td>
<td>MS-DOS 4.0</td>
<td>0 - 4 Kbyte</td>
</tr>
<tr>
<td>PS2</td>
<td>PS/2 model 80</td>
<td>Intel-80386 12.5MHz</td>
<td>MS-DOS 4.0</td>
<td>0 - 4 Kbyte</td>
</tr>
<tr>
<td>MacII</td>
<td>Macintosh II</td>
<td>Motorola-68020 16MHZ</td>
<td>A/UX 2.0 (SVR2-compatible)</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>A3000</td>
<td>Amiga 3000UX</td>
<td>Motorola-68030 25MHZ</td>
<td>AT&amp;T Unix SVR4</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>NeXTcb</td>
<td>NeXTcube</td>
<td>Motorola-68040 25MHZ</td>
<td>NeXT Mach 2.1</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>NeXTst</td>
<td>NeXTstation</td>
<td>Motorola-68040 25MHZ</td>
<td>NeXT Mach 2.1</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>SUN</td>
<td>Sun 4/390</td>
<td>SPARC CY7C601 16MIPS</td>
<td>SunOS R4.1-GFX-Rev.1</td>
<td>4 Kbyte</td>
</tr>
<tr>
<td>DEC2100</td>
<td>DEC2100</td>
<td>MIPS R2000 25MHZ 24MIPS</td>
<td>Ultrix 4.1</td>
<td>N/A</td>
</tr>
<tr>
<td>DEC5000</td>
<td>DEC 5000/200</td>
<td>MIPS R3000 25MHZ 24MIPS</td>
<td>Ultrix 4.1</td>
<td>16 Kbyte</td>
</tr>
</tbody>
</table>
The protocol specifies the protocol (e.g., TCP or UDP) used for communication. Local-addr and foreign-addr specify the IP network addresses of the local host and foreign host, respectively. Local process and foreign process are used to identify the specific processes on each system that are involved in the connection. A protocol port is used by TCP(UDP)/IP to identify the local process and the foreign process. Data received from that port are given to the associated process by TCP(UDP)/IP. The elements of the 5-tuple are specified by the system calls.

To do interprocess communication using sockets, one first creates a socket with the UNIX socket call, which specifies the communication domain, socket type, and desired communication protocol. The socket call returns a socket descriptor which the process can refer to later. This system call specifies one element of the 5-tuple, the protocol. Then the bind system call is used to assign an address to the socket. The address includes the network address for the local host and a unique protocol port number for the connection. This system call fills the second and third element of the 5-tuple. Both connection-oriented and connectionless communication use the socket and bind system calls.

For connection-oriented communication, listen is used to inform the operating system that the calling process is willing to accept a connection over a socket. Listen specifies how many connection requests can be queued by the operating system when the process executes the accept system call. Then the process executes an accept call to wait for any connection request. A connection is initiated through the connect system call, which gives the network address of the foreign host and the protocol port number identifying the connection. The connect call provides the fourth and fifth element of the 5-tuple. When the process executing the accept call receives the connection request, two way communication connection is established. Process can use UNIX read and write system calls to receive and send data, just like accessing an ordinary file. The connection is closed when either side calls close. Figure 3 shows the sequence of events that take place for connection-oriented communication [5, p. 261].
Figure 3. Socket system calls for connection-oriented communication
For connectionless communication, there are no connection set up or shut down phases. Data is sent using the `sendto` system call, which requires the address of the destination (network address of the destination host and the protocol port number) as parameters. Data is received by the `recvfrom` system call, which returns the data along with the network address of the sending process. Figure 4 shows these system calls [5, p. 262].

### 2.2.2 TLI

AT&T SVR4 supports the Transport Layer Interface (TLI), which is very similar to Berkeley sockets. TLI uses `t_open` to establish a transport endpoint. Like the `socket` system call in Berkeley sockets, `t_open` returns a file descriptor which is subsequently used by the calling process. The `t_bind` system call assigns a local address to the endpoint. Many of the structures passed between the user code and the TLI functions are structures. To simplify the dynamic allocation of these structures, the `t_alloc` system call is used. The functions of `t_listen`, `t_accept`, `t_close` are similar to those of `listen`, `accept`, and `close` in Berkeley sockets. Corresponding to `write`, `read`, `send`, and `recvfrom` in Berkeley sockets, TLI has `t_snd`, `t_recv`, `t_sndudata`, and `t_tcvudata` for data sending and receiving.

The features provided by sockets and TLI are very similar. A major difference is that TLI is designed to provide an interface to the transport layer of the OSI model and is modeled after the ISO Transport Service Definition. Therefore, TLI is a more general application program to transport layer interface. Table 2 compares sockets and TLI system calls.
Figure 4. Socket system calls for connectionless communication
Table 2. Comparison of sockets and TLI system calls

<table>
<thead>
<tr>
<th>Sockets</th>
<th>TLI</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>socket</td>
<td>t_open</td>
<td>create endpoint</td>
</tr>
<tr>
<td>bind</td>
<td>t_bind</td>
<td>bind address</td>
</tr>
<tr>
<td>listen</td>
<td>t_listen</td>
<td>specify queue</td>
</tr>
<tr>
<td>accept</td>
<td>t_accept</td>
<td>wait for connection</td>
</tr>
<tr>
<td>connect</td>
<td>t_connect</td>
<td>request a connection</td>
</tr>
<tr>
<td>write</td>
<td>t_snd</td>
<td>send data for connection-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oriented communication</td>
</tr>
<tr>
<td>read</td>
<td>t_recv</td>
<td>receive data for connection-</td>
</tr>
<tr>
<td></td>
<td></td>
<td>oriented communication</td>
</tr>
<tr>
<td>sendto</td>
<td>t_sndudata</td>
<td>send data for connectionless</td>
</tr>
<tr>
<td></td>
<td></td>
<td>communication</td>
</tr>
<tr>
<td>recvfrom</td>
<td>t_rcvudata</td>
<td>receive data for connectionless</td>
</tr>
<tr>
<td></td>
<td></td>
<td>communication</td>
</tr>
<tr>
<td>close</td>
<td>t_close</td>
<td>close a connection</td>
</tr>
<tr>
<td>none equivalent</td>
<td>t_alloc</td>
<td>allocate data structure</td>
</tr>
</tbody>
</table>
2.3 Writing Size

The number of bytes sent out with each write system call is called the writing size. In this research, the writing size is constant during each test. The five writing sizes chosen for the performance tests are 10, 100, 512, 1024, and 1460 bytes. The 10 byte size is chosen to represent very short user commands. Because a text line is around 100 bytes and listing a document and program is a frequent operation, a 100 byte writing size is also chosen. The 512 byte size is chosen because a typical disk block contains 512 bytes and file transfer often requires files sent by blocks. The receiving buffer size of some TCP/IP implementations is a multiple of 1024 bytes, so 1024 is chosen as another writing size. Each network technology places a fixed upper bound on the amount of data that can be transferred in one physical frame. Ethernet limits transfers to 1500 octets of data [4, p. 95]. The standard heading size for TCP and IP is 20 bytes. The 1460 (1500 - 2×20) byte size is chosen because that is the maximum writing size beyond which the user data must be fragmented into multiple packets. For the TCP performance tests, two more writing sizes, 1200 and 2048 bytes, are used. The 1200 byte size is chosen to get a more accurate estimation of performance trends between 1024 bytes and 1460 bytes. The 2048 byte size is chosen because it is a multiple of 1024 and is larger than the maximum packet size supported by the Ethernet. For a given experiment we send a 40 Kbytes of data using each writing size. The smallest writing size, 1 byte, is not chosen because using that writing size leads to an excessive number of datagrams so that tests in this research can not be performed in a reasonable amount of time.

It should be noted that the TCP specification does not guarantee that data from each user write is transmitted in a single segment whenever the write size does not exceed the maximum data size of a segment. A TCP implementation may group user data from several writes into a single segment to reduce the overhead of segment processing. UDP, in contrast, transforms data from each user write into one datagram and transmits the datagram before processing the next user write. User data
is not fragmented in UDP. As will be seen later, protocol segmentation of user data affects communication performance.

### 2.4 Repetition Count

Each performance experiment in this research consists of a predetermined number of trials. The number of times each experiment is repeated is called the *repetition count*, which is chosen as a compromise between the length of the experimental runs and precision of the results.

According to statistics [19], if \( X_1, X_2 \ldots X_n \) is a sequence of observations of a performance measure and \( X_i \)'s are normally distributed, then an exact \( 100(1-\alpha)\% \) confidence interval for the mean of the measure is given by

\[
\bar{X}(n) \pm t_{n-1,1-\alpha/2}\sqrt{S^2(n)/n}
\]

where \( \bar{X}(n) = \frac{\sum_{i=0}^{n} X_i}{n} \), \( t_{n-1,1-\alpha/2} \) is the upper \( 1-\alpha/2 \) critical value for t distribution with \( n-1 \) degrees of freedom and \( S^2(n) = \frac{\sum_{i=0}^{n} (X_i - \bar{X}(n))^2}{n-1} \).

The half length (i.e., \( t_{n-1,1-\alpha/2}\sqrt{S^2(n)/n} \)) is called the *absolute precision* of the confidence interval. The ratio of the confidence interval half length to the magnitude of the point estimator (i.e., \( \bar{X}(n) \)) is called the *relative precision* of the confidence interval. \( 100(1-\alpha)\% \) is called the *confidence level*.

Some preliminary tests were performed on two Amiga 3000UX computers to determine the appropriate repetition count to be used. Using 100 as the repetition count, the relative precision for the data transfer delay, a performance metric used in this research, is at least 9% with a 95% confidence level and the time required for the experiments is not excessive. Therefore, the repetition count for this research is chosen to be 100.
2.5 Artificial Host Load

Protocol performance is also measured with respect to host load. Therefore, in some experiments (see Section 3.3) an artificial load is put on the host. The host load is obtained through the use of a loading program which is a CPU intensive job. The loading program repeatedly multiplies two matrixes without sleeping. The unit of host load is defined as one copy of the program executing. Thus, a load of three units is obtained by running three copies of the loading program simultaneously.

2.6 MD-DOS/IP Package

To study the overhead of a TCP/IP implementation, we need to examine a TCP/IP implementation in detail. The MD-DOS/IP package is chosen for our study. This package was developed at the University of Maryland for IBM. It runs PS/2, PC or PC compatible machines under DOS version 3.3 or higher. The package enhances DOS so that the operating system supports memory management, timer management, and multitask scheduling, which are essential for communication software. It implements several local area network adaptor drivers including 3Com EtherLink, IBM Token Ring, Ungermann-Bass PC-NIC (used on our experimental machine), and NICps/2, etc. Also implemented are IP, UDP, TCP, BSD-socket, Sun-RPC, several application protocols including FTP, Telnet, NFS, and Rsh. For a full description of the MD-DOS/IP package, the reader is referred to [20] and [21].
2.7 PC Measurement Card

On PS/2 or PC hosts under MS-DOS, the application program can use a clock with a resolution of only 1/18 of a second which is not fine enough to accurately measure program execution. In our test, a hardware-driven microsecond-resolution measurement card [18] is used. It is possible to connect several measurement cards using an auxiliary network to form a measurement instrument. The measurement instrument offers both event-driven measurement, which generates a single log of the order and times of user-defined events on all network hosts and time-driven measurement, which simultaneously and periodically records the state of all hosts at a uniform time interval.

In our tests, the measurement card is only used to generate microsecond-resolution timestamps. The clock on the measurement card has 40 bits. The application program can easily set or read the clock on the measurement card through port I/O routines. Since each routine is equivalent to about 10 Intel-8036 assembly instructions, the cost for executing these routines is only around 40 microseconds.

2.8 Summary

Chapter 2 discussed the test environment used in this research. The hardware characteristics of the machines used for experiments in subsequent chapters were presented. This was followed by a discussion of the performance testing program. Berkeley sockets and System V TLI system calls were examined. The writing size, repetition count, and the loading program used in the tests were also discussed. Finally, the chapter discussed the MD-DOS/IP package and the PC measurement card used in the packet processing overhead tests.
Chapter 3. Performance Tests

This chapter presents results of the performance tests conducted in this research. The impact of varying processor type, varying host load, and the choice of application program interface are carefully examined.

3.1 Introduction

3.1.1 Overview of the Tests

The performance test investigates the effect the following factors have on user-perceived delay and throughput.

1. Different type of processors.

2. Load on host machine.
3. Choice of application program interface to the communication protocols.

First, Section 3.2 compares the effect of the choice of host type on transport level communication performance. Due to factors such as CPU speed, operating system, and network interface, different machines perform differently. It is useful to have a general idea about how well some typical workstations and personal computers perform in terms of network communication. The machines chosen for this research are an IBM-PS/2 model 80, IBM-compatible Dell/310, Commodore Amiga 3000UX, Apple Macintosh II, NeXTcube, NeXTstation, SUNstation 4/390, and DECstation 5000/200.

The operating system on most computers supports multitasking, which permits multiple processes to execute simultaneously. When running a communication program, it is often the case that the communication process competes with other processes for CPU cycles. Heavy CPU load slows down CPU-bound jobs dramatically. However, it is not clear how sensitive a communication program is to the CPU load. Therefore, Section 3.3 examines how sensitive the transport level transfer delay and throughput are to host load for a variety of hosts.

Section 3.4 compares different application program interfaces (APIs) to the communication protocols. The functionality of the interface between an application process and TCP (UDP) is defined in the TCP (UDP) protocol. But the protocols allow considerable freedom to implementers to design interfaces which are appropriate to a particular operating system. It is desirable to have an operating system independent interface to the communication protocol. The two most prevalent communication APIs are Berkeley sockets and the System V Transport Layer Interface [5]. Both interfaces were developed for the C language. Because of the semantics and implementation differences, different APIs yield different performance. This research compares the performance of Berkeley sockets and TLI on Amiga 3000UX which supports these two APIs.

All the tests in this research are carried out during unsocial hours to minimize the influence of other users. The test program is the only program running except for various system daemons. However,
because other hosts are connected to the Ethernet, the actual traffic on the Ethernet is not effectively controlled. To assess the representativeness of the tests results, one version of the performance testing program, TCP with sockets, was run between the two A3000s on three consecutive days. Using a writing size of 1460 bytes, the testing program transmits 40 Kbytes of data 100 times. The highest average transfer delay obtained from the three day experiments is 10% higher than the slowest.

In all the tests, the connection establish and close phases are not considered. All data presented is measured from the data transfer phase, after connections are established and before they are closed.

3.1.2 Performance Metrics

Two performance metrics are used in this research.

1. Data transfer delay

The data transfer delay is defined as the time that elapses from the instant the receiving process receives the first packet until the instant the recipient receives the last packet [8]. The data transfer delay, recorded by the user process, includes the time it takes to process the second and the subsequent packets by the network software. To account for the processing time for the first packet, it is necessary to record the beginning time when the first packet is processed by the network software. This is impossible for the user process which does not know what happens in the underlying network software. When a sufficient number of packets are transferred, which is the case in this research, this inaccuracy is negligible.

2. Throughput

Throughput is defined as the amount of user data transferred per unit time. In this research, it is the total number of bytes transferred divided by the data transfer delay.
3.2 Sensitivity of User-level Performance to Choice of Host

3.2.1 Test Procedure

The performance testing program described in Section 2.2 is run on six pairs of sending and receiving hosts: PS2 to D310, MacII to MacII, A3000 to A3000, NeXTcb to NeXTst, SUN to SUN, and DEC5000 to DEC5000. Five writing sizes (10, 100, 512, 1024, and 1460 byte) are used for both the TCP and UDP performance tests. The TCP tests use two more writing sizes, 1200 and 2048 bytes. Each test involves transmitting 40 Kbytes of data from the sending host to the receiving host using TCP or UDP. In the test, the sending program is a tight loop that repeatedly calls write to achieve the maximum user transmission rate. Each test is repeated 100 times on each machine pair. Each trial uses a unique port number to minimize the effect that one trial has on another (i.e., trial n does not reuse buffers allocated in trial n-1). The data transfer delay is recorded in each trial. The results are averaged over 100 trials.

3.2.2 Test Results and Observations

Transmission Control Protocol (TCP)

Table 3 lists the mean data transfer delay over 100 trials with TCP for each machine pair. Also listed is the half length of the confidence interval for a 95% confidence level. When an attempt was made to write 2048 bytes with one write call on the PS2, an error was returned. So the data transfer delay for PS2 to Dell310 at that writing size is not available. Based on the data in Table 3, Figure 5 depicts the TCP throughput achieved on different machine pairs.
Table 3. The data transfer delay to send 40 Kbytes on TCP (numbers represent the sample mean and, in parenthesis, the half length of the confidence interval with 95% confidence level)

<table>
<thead>
<tr>
<th>writing size (byte)</th>
<th>PS2 to D310 (msec)</th>
<th>MacII to MacII (msec)</th>
<th>A3000 to A3000 (msec)</th>
<th>NeXTcb to NeXTst (msec)</th>
<th>SUN to SUN (msec)</th>
<th>DEC5000 to DEC5000 (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>2617.9 (7.2)</td>
<td>3015 (12.6)</td>
<td>4041 (13.5)</td>
<td>579 (12.2)</td>
<td>887 (8.3)</td>
<td>560 (8.5)</td>
</tr>
<tr>
<td>100</td>
<td>606.4 (27.4)</td>
<td>501 (10.8)</td>
<td>594 (6.3)</td>
<td>185 (9.1)</td>
<td>130 (1.2)</td>
<td>83.7 (1.3)</td>
</tr>
<tr>
<td>512</td>
<td>357.7 (16.8)</td>
<td>266 (8.3)</td>
<td>216 (1.0)</td>
<td>90 (8.1)</td>
<td>70 (0)</td>
<td>57.0 (0.1)</td>
</tr>
<tr>
<td>1024</td>
<td>320.2 (8.6)</td>
<td>216 (1.0)</td>
<td>174 (3.0)</td>
<td>58 (0.4)</td>
<td>63 (1.0)</td>
<td>39.6 (1.3)</td>
</tr>
<tr>
<td>1200</td>
<td>329.0 (4.9)</td>
<td>282 (9.9)</td>
<td>257 (12.2)</td>
<td>123 (9.4)</td>
<td>73 (3.7)</td>
<td>38.8 (3.2)</td>
</tr>
<tr>
<td>1460</td>
<td>346.1 (16.4)</td>
<td>292 (12.2)</td>
<td>254 (14.8)</td>
<td>167 (12.5)</td>
<td>57 (1.7)</td>
<td>39.3 (1.3)</td>
</tr>
<tr>
<td>2048</td>
<td>*</td>
<td>237 (4.3)</td>
<td>175 (1.8)</td>
<td>59 (0.5)</td>
<td>56.9 (2.0)</td>
<td>38.9 (4.7)</td>
</tr>
</tbody>
</table>

* PS/2 crashed with write size of 2048 bytes
Figure 5. TCP throughput on different machine pairs
As expected, the data transfer delay decreases as the CPU speed increases as shown in Table 3. A less obvious factor which affects the data transfer performance is the TCP receiving buffer size. The DECstation allocates a 16 K receiving buffer for each TCP connection while the receiving buffer size on MacII, A3000, NeXT, and SUN is 4K. The TCP receiving buffer size on D310 can be specified by the users in the range of 0 to 4 K. In this test, the size on D310 is 2 K (see Figure 17 in Chapter 4 for other experiments with other buffer sizes). In TCP, the receiver controls the amount of data sent by the sender. Based on the size of the available receiving buffer, the receiver returns an advised window on every acknowledgement. The window indicates an allowed amount of data that a sender can transmit before receiving new permission. Therefore, a smaller receiving buffer means the sender has to stop more frequently, resulting in longer transfer delays. So the different TCP receiving buffer sizes on these machine pairs is another factor contributing to the delay disparity among these machines.

Table 3 shows that when the writing size is less than 1024 bytes, the data transfer delay on all machine pairs decreases as writing size increases. In ordinary data transfer, TCP waits to transmit a data segment until it has completely filled a segment or until the segment flush timeout expires. When the segment flush timeout expires, TCP sends the segment whether it is full or not. A larger writing size means the TCP data segment has a larger size when data is flushed. Since the total data transferred is fixed at 40 Kbytes, a larger data segment size implies fewer segments are sent, which decreases the segment processing overhead and thus reduces the network delay. A large writing size also decreases the user process overhead by reducing the number of write system calls needed for the data transfer.

When the writing size is greater than 1024 bytes, Table 3 shows that, except for the DECstation, a writing size which is a multiple of 1024 byte achieves better performance. This is due to the TCP flow control mechanism. The sending TCP has a sending window which controls how much data it can send before it must wait for an acknowledgement from the receiver. The sending window size is decided by the advised window sent back by the receiving TCP and based on the size of its available receiving buffer. The receiving buffer size on the D310 (2K), and on the A3000, MacII,
NeXT, and SUN (all 4K), is a multiple of 1024 bytes. So after sending a multiple of 1 Kbyte data (2 K or 4 K), the sender must stop to wait for the acknowledgement. When using a writing size which exceeds but is not a multiple of 1024 bytes, user data from some writes is broken into two parts by TCP. The first part is sent out immediately by TCP while the remaining part stays in the TCP sending buffer because TCP has used up its sending window. The user data fragmentation increases the total number of transferred segments and hence results in longer delay. When using a writing size which is a multiple of 1024 bytes, Fewer user data fragmentations occurs since user data from different writes can fit in one window.

For the DECstation, the delay is around 39 msec when the writing size is greater than 1024 bytes. This delay translates to a throughput of 8.4 Mbps. Since the Ethernet used in this test has a throughput of 10Mbps, the DECstation data transfer throughput appears to be limited by the physical network throughput. This conjecture is supported by the observation in Table 3 that the delay is almost constant for different writing sizes equal or larger than 1024 bytes.

User Datagram Protocol (UDP)

The results of measured UDP transfer delay are tabulated in Table 4. Numbers in Table 4 represent the mean of the delay over 100 trials, the half length of the confidence interval with 95% confidence level, and the data loss rate (amount of lost data divided by the total amount of transferred data). Based on the data in Table 4, Figure 6 depicts the UDP throughput achieved on different machine pairs (PS2 to D310 is not included because the loss rate on that pair is too high when the writing size is greater than 512 bytes).

Table 4 shows that on the A3000, MacII, NeXT, SUN, and DECstation, the delay always decreases as the writing size increases, which is different from the behavior for TCP (see Table 3). Two explanations account for this. First, unlike TCP, UDP has no flow control; therefore outgoing data never waits for acknowledgement of previously sent data. Second, because the writing size in this
Table 4. The data transfer delay to send 40 Kbytes on UDP (numbers represent the sample mean, and, in parentheses, the half length of the confidence interval with 95% confidence level)

<table>
<thead>
<tr>
<th>Writing size</th>
<th>PS2 to D310 delay (msec)</th>
<th>PS2 to D310 loss rate (%)</th>
<th>MacII to MacII delay (msec)</th>
<th>MacII to MacII loss rate (%)</th>
<th>A3000 to A3000 delay (msec)</th>
<th>A3000 to A3000 loss rate (%)</th>
<th>NeXTc to NeXTst delay (msec)</th>
<th>NeXTc to NeXTst loss rate (%)</th>
<th>SUN to SUN delay (msec)</th>
<th>SUN to SUN loss rate (%)</th>
<th>DEC5000 to DEC5000 delay (msec)</th>
<th>DEC5000 to DEC5000 loss rate (%)</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>6787 (1.8)</td>
<td>0%</td>
<td>8871 (24.5)</td>
<td>0.01%</td>
<td>11920 (14.3)</td>
<td>0.57%</td>
<td>3359 (1.9)</td>
<td>0.05%</td>
<td>1967 (2.0)</td>
<td>0.07%</td>
<td>1741 (8.3)</td>
<td>3.7%</td>
</tr>
<tr>
<td>100</td>
<td>718 (0.4)</td>
<td>0%</td>
<td>1021 (7.2)</td>
<td>0%</td>
<td>1317 (8.2)</td>
<td>0%</td>
<td>356 (0.6)</td>
<td>0.61%</td>
<td>208 (1.7)</td>
<td>0.32%</td>
<td>176 (0.3)</td>
<td>0.03%</td>
</tr>
<tr>
<td>512</td>
<td>171 (0.4)</td>
<td>1.8%</td>
<td>272 (3.4)</td>
<td>0.01%</td>
<td>298 (0.7)</td>
<td>0%</td>
<td>82 (0.3)</td>
<td>0.05%</td>
<td>49 (0.2)</td>
<td>0%</td>
<td>51 (0.3)</td>
<td>0.09%</td>
</tr>
<tr>
<td>1024</td>
<td>104 (0.4)</td>
<td>20.7%</td>
<td>181 (3.1)</td>
<td>0%</td>
<td>166 (0.6)</td>
<td>0%</td>
<td>43 (0.3)</td>
<td>1.5%</td>
<td>40 (0)</td>
<td>0%</td>
<td>34 (0.3)</td>
<td>0.56%</td>
</tr>
<tr>
<td>1460</td>
<td>87.6 (0.8)</td>
<td>57.0%</td>
<td>168 (1.2)</td>
<td>0.1%</td>
<td>139 (1.2)</td>
<td>0%</td>
<td>38 (0.3)</td>
<td>0.17%</td>
<td>39 (0.4)</td>
<td>0%</td>
<td>34 (0.2)</td>
<td>0.07%</td>
</tr>
</tbody>
</table>
Figure 6. UDP throughput on different machine pairs
test is smaller than the Ethernet maximum packet size, the user data from each write is encapsulated in a single datagram by UDP and immediately sent out. Therefore, delay decreases as writing size increases because a larger writing size requires fewer datagrams to be transmitted.

Since UDP has no flow control, the data transfer speed is only limited by the packet processing speed on the host and the transmission medium speed. Therefore, the CPU speed has a profound influence on the UDP data transfer delay. As shown in Table 4, the NeXT, SUNstation, and DECstation outperform the MacII and A3000 by a factor of at least 2 to 3 times for all writing sizes.

The measured results for PS2 to D310 shows the disadvantage of UDP. The user data loss rate jumps when the writing size exceeds 512 bytes. With a 1460 byte writing size, more than half the user data is lost during transmission because UDP does not have flow control. In our experiment, PS2 serves as the sender while D310 is receiver. Since PS2 is faster than D310, the sender overwhelms the receiver. This contrasts with the other UDP experiments, which involve peers that are identical except for main memory size. When using UDP, care must be taken to ensure that the sending speed matches the receiving side processing speed.

Figures 7 and 8 compare the results of Table 3 and Table 4. In comparing the delay for TCP and UDP on MacII, A3000, NeXT, SUN, and DECstation (see Figures 7 and 8), it is noted that when the writing size is small, TCP performs better than UDP but as the writing size increases, UDP outperforms TCP. This phenomenon is due to the protocol differences between TCP and UDP. TCP offers connection-oriented reliable data transfer which means TCP views user data as a stream. Its task is to ensure the receiver gets the same stream that sender sends out. To improve the transmission efficiency, TCP does not send out a data segment immediately after it receives user data, instead it waits for the user to fill the segment. Unlike TCP, UDP provides connectionless data delivery service. Data received from the user is immediately sent out as a datagram. So UDP transmits more packets than TCP does when the user chooses a small writing size. Thus TCP yields higher throughput than UDP when the writing size is small. On the other hand, TCP has a flow control mechanism which UDP does not have. When the sender sends out data too fast, TCP slows
the sender down. At larger writing sizes, the user can transmit data at a rate too high for the receiver. Thus TCP's flow control mechanism comes into the picture when the writing size becomes large, and explains why UDP yields higher throughput than TCP when the writing size is large.

It is up to the user to choose the underlying communication protocol for his or her application. The test results suggest that when a small writing size is used for the application, TCP achieves both higher throughput and reliability, and is the method of choice. But when the writing size is large, UDP should be used to achieve the highest throughput except if the sender is much faster than the receiver, while TCP should be used for reliability.

3.3 Sensitivity of User-level Performance to Host Load

3.3.1 Test Procedure

The results of this section are obtained by running the performance testing program on two machine pairs: MacII to MacII and A3000 to A3000. The matrix multiplication loading program described in Section 2.5 is used to generate an artificial workload. The loading program runs as a background process when the test is carried out. Both the sending host and the receiving host run the same loading program. Three cases are compared by running either zero, one copy or three copies of the loading program. The preceding section contains the zero-copy case results, which are repeated in the tables of this section to simplify comparison.

Five writing sizes (10, 100, 512, 1024, and 1460 bytes) are used and a total of 40 Kbytes of data are sent using TCP or UDP. For each writing size, the test is repeated 100 times on each machine pair for each loaded case. The results are obtained by averaging over 100 trials.
Figure 7. Comparison of throughput for TCP and UDP (on MacII and A3090)
Figure 8. Comparison of throughput for TCP and UDP (on NeXT, SUN, and DECstation)
3.3.2 Test Results and Observations

Tables 5 and 6 present the data transfer delay for different host loads using TCP and UDP, respectively. The numbers in Tables 5 and 6 represent the average transfer delay over 100 trials and the half length of the confidence interval with a 95% confidence level.

In TCP (see Table 5), when the writing size is 10 bytes, the transfer delay increases significantly as the host load increases. For both the MacII and A3000 host, the delay triples when the host load increases from one unit to three units. This is also true for the MacII, but not the A3000 host, as the writing size increases to 100 bytes. For the other cases, the delay is not very sensitive to the host load. The delay increases by at most 23% when host load triples.

The UDP case (see Table 6) is very similar to TCP. When the writing size is greater than 100 bytes on the MacII and 10 bytes on the A3000, the host load does not have much influence on the data transfer delay. In the UDP case, the delay roughly quadruples at 10 bytes when the host load triples.

The TCP and UDP test results need to be examined carefully. First, the test results do indicate that host load has a profound impact on the transfer delay when the writing size is 10 bytes. In our tests, a total of 40 Kbytes of data are transferred. With a 10 byte write size, UDP transmits 4096 datagrams while TCP transmits at most 4096 segments. To process this amount of packets on either host requires a large amount of CPU time. Thus, as the host load rises, the transfer delay increases dramatically. When the writing size increases to 512 bytes, to transfer 40 Kbytes of data, UDP only sends out 80 datagrams and TCP transmits at most 80 segments. Because less CPU time is needed to process 80 packets than to process 4096 packets, host load barely affects the data transfer delay at a write size of 512 bytes, but dramatically affects delay at a write size of 10 bytes in our test.
Table 5. The data transfer delay to send 40 Kbytes on TCP for varying host load (numbers represent the sample mean and, in parentheses, the half length of the confidence interval with 95% confidence level)

<table>
<thead>
<tr>
<th>Writing Size</th>
<th>\textbf{MacII to MacII} \begin{tabular}{c} (msec) \end{tabular}</th>
<th>\textbf{A3000 to A3000} \begin{tabular}{c} (msec) \end{tabular}</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>\begin{tabular}{c} load=0 \end{tabular}</td>
<td>\begin{tabular}{c} load=1 \end{tabular}</td>
</tr>
<tr>
<td>10</td>
<td>3015 \begin{tabular}{c} (12.6) \end{tabular}</td>
<td>5480 \begin{tabular}{c} (22.7) \end{tabular}</td>
</tr>
<tr>
<td>100</td>
<td>501 \begin{tabular}{c} (10.8) \end{tabular}</td>
<td>507 \begin{tabular}{c} (12.7) \end{tabular}</td>
</tr>
<tr>
<td>512</td>
<td>266 \begin{tabular}{c} (8.3) \end{tabular}</td>
<td>319 \begin{tabular}{c} (12.9) \end{tabular}</td>
</tr>
<tr>
<td>1024</td>
<td>209 \begin{tabular}{c} (2.7) \end{tabular}</td>
<td>220 \begin{tabular}{c} (3.4) \end{tabular}</td>
</tr>
<tr>
<td>1460</td>
<td>292 \begin{tabular}{c} (12.2) \end{tabular}</td>
<td>324 \begin{tabular}{c} (17.6) \end{tabular}</td>
</tr>
</tbody>
</table>
Table 6. The data transfer delay to send 40 Kbytes on UDP for varying host load (numbers represent the mean and, in parentheses, the half length of the confidence interval with 95% confidence level)

<table>
<thead>
<tr>
<th>Writing Size (byte)</th>
<th>MacII to MacII (msec)</th>
<th>A3000 to A3000 (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>load=0</td>
<td>load=1</td>
</tr>
<tr>
<td>10</td>
<td>8871</td>
<td>17585</td>
</tr>
<tr>
<td></td>
<td>(24.5)</td>
<td>(32.3)</td>
</tr>
<tr>
<td>100</td>
<td>1021</td>
<td>1443</td>
</tr>
<tr>
<td></td>
<td>(7.2)</td>
<td>(15.4)</td>
</tr>
<tr>
<td>512</td>
<td>272</td>
<td>283</td>
</tr>
<tr>
<td></td>
<td>(3.4)</td>
<td>(3.5)</td>
</tr>
<tr>
<td>1024</td>
<td>181</td>
<td>197</td>
</tr>
<tr>
<td></td>
<td>(3.1)</td>
<td>(3.1)</td>
</tr>
<tr>
<td>1460</td>
<td>168</td>
<td>172</td>
</tr>
<tr>
<td></td>
<td>(1.2)</td>
<td>(4.6)</td>
</tr>
</tbody>
</table>
These results suggest that as the total number of transmitted packets during a fixed time interval increases, transfer delay becomes increasingly sensitive to host load no matter what writing size is used. To verify this conjecture, additional tests are reported in Table 7 in which the amount of data sent is 512,000 bytes and the writing size is 512. The tests were carried out on two MacIIs.

To transfer 512,000 bytes using 512 as the writing size, UDP transmits 1000 datagrams while TCP transfers at most 1000 segments. As explained earlier, the transfer delay in the new tests should be very sensitive to the host load changes which is exactly what Table 7 indicates. In the new tests, the delay more than doubles as the host load increases from one unit to three units for both TCP and UDP.

Tables 5, 6, and 7 suggest that the user-perceived network delay becomes more sensitive to the host load as more data packets are generated by the network software during a fixed time interval. To reduce the influence of host load on communication programs, it is helpful to minimize the number of packets generated by the underlying network software for the user program. One strategy that the user process can follow is to choose a writing size that is as large as possible, but does not exceed the maximum packet size supported by the network. If TCP is used, another strategy that a user can follow is to increase the segment flush timeout period so that more data is grouped into one segment, hence reducing the total number of segments transferred.
Table 7. The data transfer delay for 512,000 bytes between two MacIIs (numbers represent the sample mean and, in parentheses, the half length of the confidence interval with 95% confidence level)

<table>
<thead>
<tr>
<th>Protocol</th>
<th>load=0 (msec)</th>
<th>load=1 (msec)</th>
<th>load=3 (msec)</th>
</tr>
</thead>
<tbody>
<tr>
<td>TCP</td>
<td>3086</td>
<td>4806</td>
<td>8429</td>
</tr>
<tr>
<td></td>
<td>(70.4)</td>
<td>(93.3)</td>
<td>(301.6)</td>
</tr>
<tr>
<td>UDP</td>
<td>3533</td>
<td>6524</td>
<td>13775</td>
</tr>
<tr>
<td></td>
<td>(60)</td>
<td>(40.6)</td>
<td>(196.9)</td>
</tr>
</tbody>
</table>
3.4 Sensitivity of User-level Performance to Choice of Application Program Interface

3.4.1 Test Procedure

Four versions of the performance testing program have been run on the A3000. They are TCP with sockets, UDP with sockets, TCP with TLI, and UDP with TLI. The same five writing sizes chosen in the previous tests are used for this test. Each version of the testing program is repeated 100 times for each writing size. The total amount of data transferred in each testing program run is 40 Kbytes. The results are averaged over 100 trials.

3.4.2 Tests Results and Observations

Table 8 reports the data transfer delay for System V TLI on A3000, and compares it to the delay for Berkeley sockets reported in Tables 5 and 6. Based on Table 8, Figure 9 compares the throughput for sockets and TLI.

Transmission Control Protocol (TCP)

As Table 8 shows sockets performs better than TLI for various writing sizes. With a writing size of 100 bytes, the transfer delay of sockets is 24% less than that of TLI. Such great delay disparity between these two application program interfaces is unexpected. Obviously, the performance difference is due to the difference between the sockets and TLI implementations. Without the source code for these implementations, the difference can not be explained.
Table 8. The data transfer delay for Berkeley sockets and System V TLI between A3000 and A3000 (numbers represent the sample mean and, in parentheses, the half length of the confidence interval with 95% confidence level)

<table>
<thead>
<tr>
<th>Writing Size</th>
<th>TCP (msec) Sockets</th>
<th>TLI</th>
<th>UDP (msec) Sockets</th>
<th>TLI</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>4041 (13.5)</td>
<td>5100 (10.0)</td>
<td>11920 (14.3)</td>
<td>9852 (20.1)</td>
</tr>
<tr>
<td>100</td>
<td>594 (6.3)</td>
<td>782 (14.7)</td>
<td>1317 (8.2)</td>
<td>1072 (3.1)</td>
</tr>
<tr>
<td>512</td>
<td>216 (1.0)</td>
<td>257 (7.7)</td>
<td>298 (0.7)</td>
<td>236 (1.4)</td>
</tr>
<tr>
<td>1024</td>
<td>174 (3.0)</td>
<td>216 (1.5)</td>
<td>166 (0.6)</td>
<td>148 (0.9)</td>
</tr>
<tr>
<td>1460</td>
<td>254 (14.8)</td>
<td>255 (1.8)</td>
<td>139 (1.2)</td>
<td>115 (1.0)</td>
</tr>
</tbody>
</table>

User Datagram Protocol (UDP)

Contrary to the situation in TCP, TLI outperforms sockets when UDP is the underlying protocol (see Table 8). For UDP, the greatest delay disparity occurs when the writing size is 512 bytes. At this point, sockets uses 20.8% more time than TLI.
Figure 9. Comparison of throughput for sockets and TLI on A3000
From the experiments, it can be concluded that application program interface does affect the user-perceived communication performance. Sometimes this effect is not negligible. While it is essential to improve the performance of the network and the network software, it is also important to improve the performance of the application program interface. After all, the user can only access network services through an application program interface.

3.5 Summary

This chapter presented the results of the performance tests. The testing environment was described. In the first part, the effects of different processors on the communication program were examined. In the second part, the impacts of host load on the user-perceived network delay were studied. Finally, this chapter discussed the effects of application program interface on the communication program.
Chapter 4. TCP/IP Overhead Study

This chapter analyzes TCP/IP in the MD-DOS/IP package. A direct system measurement technique is used in this research. Section 4.1 describes the internal structure of the TCP/IP implementation and the instrumentation of TCP/IP. Section 4.2 discusses the overhead for packet processing in TCP/IP. Based on the analysis, sources of the packet processing overhead are identified. The implications of the overhead study results are also discussed. A detailed analysis of the TCP implementation is presented in Section 4.3. Implementation suggestions are made based on the analysis results. Section 4.4 studies the bottleneck of data communication using TCP/IP.

4.1 Instrument the TCP/IP Implementation

4.1.1 The Internal Structure of the TCP/IP Implementation

The MD-DOS/IP package [20,21] was chosen for the overhead study. Figure 10 shows the internal structure of the TCP, IP, and network driver of the package.
Figure 10. Internal structure of the TCP/IP implementation
The MD-DOS/IP package will be described by tracing through the calls made in the sending and receiving paths. Figure 11 presents the calling hierarchy for a send request for TCP/IP. When a user process wishes to send out data, it calls write(). From the user's point of view, the write() system call sends out the user data. Actually, the user data presented in the write() call is copied to the TCP sending buffer. The data copy is accomplished by the TCP routine tcp_write() invoked by write(). It should be noted that if the TCP sending buffer is full, tcp_write() and consequently, the calling routine, write(), may block. The TCP task tcp_timers() checks the TCP sending buffer. If tcp_timers() finds that there is data to send (the user has filled a segment or the segment flush timeout expires) and the sending window is not closed, it calls snd_data() to transmit a data segment. The snd_data() routine decides how much data to send in the data segment according to the sending window size and the amount of available data in the TCP sending buffer. It also starts a timer for the data segment. The TCP routine snd_normal() is called by snd_data() to fill the necessary header fields of the data segment. The snd_normal() routine then calls transmit() to compute the data segment checksum (using tcp_cksum()) and pass the data segment to IP layer. The IP routine ip_send() is called by transmit() with the data segment as a parameter. If the data segment size plus the standard IP header size exceeds the maximum packet size supported by the underlying network, the IP routine ip_grenade() breaks the data segment into fragments to fit into the network packet. IP prepends an IP header and any IP options to the data segment or the data segment fragment to generate a datagram. The resulting IP datagram is then given to the network driver. The driver routine h_send() is called by ip_send() to send out the datagram. The h_send() routine is responsible for mapping the datagram’s IP address to the network physical address and copying the final packets to the network via an Ungermann-Bass PC-NIC Ethernet adapter.

Figure 12 presents the TCP/IP calling hierarchy for receiving a packet. When the Ethernet card receives data from the network, it generates an interrupt. The interrupt handler h_interrupt() copies data off the network board into an mbuff chain. Then it looks for a listener for the incoming packet. If IP is the listener, which is the case in our experiments, h_interrupt() puts the mbuff chain on IP’s queue, wakes up the IP task ip_intrsvc() and returns. When the IP task gets awakened, it takes the
Figure 11. TCP/IP calling hierarchy for sending a packet
packet off its queue and calls ip_demux() to process the packet. The ip_demux() routine first verifies the header of the packet, then it checks to see if the packet is for the local host. If the packet is not for the local host, it routes the packet through another interface. If it is, ip_demux() determines which transport layer protocol the packet is intended for and upcalls the protocol. The TCP routine tcp_recv() is upcalled by the IP task to process the incoming segment. The tcp_recv() routine checksums the incoming segment (using tcp_cksum()) and chains the user data in the incoming segment to a TCP assembly line (see Figure 10). The TCP task tcp_timers() checks the TCP assembly line and copies the available data in the assembly line to the user buffer. The user process gets the data through the read() system call.

4.1.2 Modifications to TCP, IP and Driver Routines

This section describes modifications made to the MD-DOS/IP software to instrument it. The dashed lines in Figure 10 represent the interfaces between the network software entities. The TCP, IP, and network driver routines are modified so that when a packet crosses an interface, a log entry is generated and appended to a data trace. Each log entry contains the timestamp obtained from a microsecond-resolution measurement card [18] mounted on the host. Log entries are appended to the memory area as the protocol executes.

Packets from user to TCP

The tcp_write() routine copies the user data to the TCP sending buffer. This routine is modified to append two log entries. One timestamp is obtained immediately after the routine begins; the other is obtained just before the routine returns. The TCP routine tcp_timers() checks TCP states (e.g., timeout, sending and receiving buffer) and takes appropriate action. The TCP routine snd_data() is called by tcp_timers() whenever it wishes to send out a data segment. The snd_data() routine decides how much data can be sent and starts a timer for the data segment. It
Figure 12. TCP/IP calling hierarchy for receiving a packet
then calls other TCP routines to append the TCP header to the data, computes the checksum, and passes the segment to the IP layer. The TCP routine tcp_timers() is modified to record the timestamps obtained before and after snd_data() is called.

Packets from TCP to IP

The TCP routine snd_data() eventually calls the TCP routine transmit() to send out the segment. The transmit() routine passes the TCP segment to IP through the IP routine ip_send(). A pointer to the TCP segment is among the parameters with which transmit() calls ip_send(). The ip_send() routine converts the segment into a datagram and sends the datagram onto the network. The transmit() routine is modified to record two timestamps obtained before and after ip_send() is called.

Packets from IP to network interface

IP receives a segment from TCP through the ip_send() routine. After processing the packet and deciding on routing issues, ip_send() calls ip_grenade(). The main task of ip_grenade() is datagram fragmentation. If the size of the datagram exceeds the maximum packet size, ip_grenade() breaks the datagram into fragmentations. The ip_grenade() routine calls the driver routine h_send() to send out the resulting datagram. The h_send() routine changes the IP address of the datagram to the network physical address and copies the datagram onto the network. Timestamps are recorded before and after h_send() is called in ip_grenade().

Packets from network interface to IP

When the Ethernet card receives data, it generates an interrupt. The interrupt handler copies the packet off the network board and wakes the corresponding task. The Ungermann-Bass network adapter interrupt handler h_interrupt() is modified to record the time at the beginning of the routine. The IP task ip_intrsvc() is awakened to process the incoming packet. The ip_intrsvc() routine
is modified to append a log entry at this stage. The timestamp for this entry is taken at the beginning of the routine. This marks the beginning of the IP process.

Packets from IP to TCP

The IP routine \texttt{in\_intrsvc()} calls \texttt{ip\_demux()} which verifies the header of the incoming packet and demultiplexes the packet. Routine \texttt{ip\_demux()} determines whether the packet is destined for the local host. If that is the case, it upcalls to the intended upper layer protocol. This routine is modified to record the timestamps taken before and after the upcall.

Packets from TCP to user

The TCP routine \texttt{tcp\_recv()} is called when the \texttt{ip\_demux()} routine decides the packet is for the TCP protocol. Routine \texttt{tcp\_recv()} processes the incoming segment and chains the user data from the segment to the TCP assembly line. TCP task \texttt{tcp\_timers()} is responsible for copying the data from the TCP assembly line to the user buffer. TCP task \texttt{tcp\_timers()} is modified to record the timestamps taken before and after the data copy.

Time spent on TCP checksum and timer management are also recorded. The TCP checksum routine \texttt{tcp\_cksum()} is called by the \texttt{transmit()} routine when TCP transmits a segment and by the \texttt{tcp\_recv()} routine when TCP receives a segment. Both \texttt{transmit()} and \texttt{tcp\_recv()} are modified to record the timestamp obtained before and after \texttt{tcp\_cksum()} is called. The timer management starts a timer when a data segment is sent out and updates the retransmission timeout period and sending window when an acknowledgement is received. The TCP routine \texttt{snd\_data()} calls \texttt{start()} each time a data segment is transmitted to start a timer. The \texttt{snd\_start()} routine is modified to record the timestamp taken before and after \texttt{start()} is called. The TCP routine \texttt{acknowledgement()} is for managing the retransmission timeout period and sending window. Jacobson's slow-start and round-trip-time variance estimation algorithms [14] are used in this piece of code. Routine \texttt{ac-}
is modified to record the timestamps obtained immediately after the routine begins and just before it returns.

During the experiments, each log entry generated is appended to the trace. Storing log entries in memory causes less overhead than appending log entries to a disk file during the experiment. To reduce the overhead introduced by log entry recording, all log entries are written to main memory during the experiment. The data trace is copied to disk when data transfer is over and TCP sends out a FIN segment to close the connection. In our experiments, the memory available to the TCP/IP process is large enough to hold the data traces. If the data trace generated by the experiment exceeds the size of the available memory, the data trace needs to be transferred to the disk periodically.

4.2 The Overhead of TCP/IP Packet Processing

This section addresses the overhead of running the TCP, IP, network driver, and the user application program. The TCP/IP implementation of the MD-DOS/IP package is instrumented as described in Section 4.1.2. By analyzing the data traces obtained, the sources of overhead in TCP/IP packet processing are identified.

4.2.1 Test Procedure

The tests involve a PS/2 and a Dell/310 system. Both machines run MS-DOS 4.0 with the MD-DOS/IP package providing the network services. The TCP, IP, and network driver routines of the MD-DOS/IP package are modified as described in Section 4.1.2. The modified TCP/IP is
recompiled to produce the instrumented version of TCP/IP. The instrumented version of TCP/IP is only run on the Dell/310, which is equipped with the measurement card.

The tests are carried out by running the sockets version TCP performance testing program described in Section 2.2 between PS2 and D310. Each test involves establishing a TCP connection, transferring 40 Kbytes of data, and closing the connection. The instrumented TCP/IP records the time spent in each software entity during the data transfer. The writing size is chosen to be 1024 which yields the best network performance as shown in Table 3.

4.2.2 Test Results and Observations

Figure 13 shows what happens inside the network software in a single trial on the sending side during the data transfer.

The horizontal line represents the time. The vertical line represents the system states. There are six system states.

1. The system is idle.

2. The network driver is processing the packet.

3. IP is processing the packet.

4. TCP is processing the packet, but TCP is not copying data from the user buffer to the TCP sending buffer.

5. TCP is copying data from the user buffer to the TCP sending buffer.

6. The user is processing the packet.
Figure 13. The behavior of TCP/IP on the sending side (D310 to PS2)
At time 0, the sending window size is 2K, and the TCP sending buffer (see Figure 10) size is 4K. Figure 13 shows that before time 2000, two chunks (1 Kbyte each) of user data are copied to the TCP sending buffer. Then, between time 2120 and 5302, TCP, IP, and the driver are busy sending out the first data segment (1460 bytes). This is followed by more user data (1 Kbyte) being copied to the TCP buffer and TCP, IP, and the driver become busy sending out the second data segment (588 bytes). This procedure occurs repeatedly throughout the data transfer phase. At time 8607, the system becomes idle. The user process is blocked at that time because the TCP sending buffer is full. The size of the TCP sending buffer is 4K. By time 8607, the user has already copied 4K of data to the TCP buffer. TCP keeps the data in its sending buffer until the data is acknowledged. At time 8607, TCP, IP, and the driver will not become busy either. This is because TCP has used up its sending window (1460 bytes + 588 bytes = 2 Kbytes) by that time and must wait for an acknowledgement before it can send data again. The system stays idle until time 24,053 when an acknowledgement is received by the driver. Then the acknowledgement is processed and at time 24,529, TCP, IP, and the driver become busy sending out another data segment (1460 bytes). The data transfer continues with the repetition of similar events.

Note in Figure 13, the system is idle from time 8607 to time 24,053. The reason for this long period of idle time is that the TCP sending buffer is full and the sending TCP has used up its sending window. More tests are carried out to measure the system idle time due to this reason. The socket version of the TCP performance testing program is run between the PS2 and D310 with the sending program on D310. Each test involves establishing a TCP connection, transmitting 40 Kbytes of data using 1024 as the writing size, and closing the TCP connection. The instrumented version of TCP/IP on D310 generates the traces of the data transmission on the sending side. By analyzing the traces, the system idle time is obtained. The tests results are average over 30 trials. Table 9 shows the results. The total time in Table 9 is the time that elapses from the instant the sending program first calls write to transfer data until the instant the sender finishes processing the last acknowledgement. The idle time in Table 9 is the total system idle time (because the TCP sending buffer is full and the sending TCP has used up its sending window) during the data transfer test.
Table 9 shows that when using the instrumented version of TCP/IP, it takes the sender 344.38 milliseconds (the elapsed time between the user sends the first packet and the TCP has processed the last acknowledgement) to transfer 40 Kbyte data, while on the original TCP/IP, the average transfer delay is 320.2 milliseconds (see Table 3). The instrumentation overhead comes mainly from the clock reading cost. For each test, the clock is read fewer than 260 times with the cost for each clock reading being around 40 microseconds (See Section 2.7). Thus, the instrumentation overhead for each test is around $260 \times 40$ microseconds $= 10.4$ milliseconds. Compared with the total time in Table 9, this overhead is negligible.

The data in Table 9 indicates the system idle time is 156.491 milliseconds, which means the system is idle at least 45.3% of the time during the data transfer. There are three tasks running on the sender (see Figure 10). The first one is the acknowledgement processing task which is invoked when the sender receives an acknowledgement from the receiver. The second one is the user task which calls TCP routine `tcp_write()` (see Figure 11) to copy the user data to the TCP sending buffer. The third task is responsible for processing the user data and sending out the resulting packet onto the network. The sender has no control over when to run the first task, since the task is invoked by the arrival of an acknowledgement. The user task is blocked when the TCP sending buffer is full. The third task is blocked when TCP has used up the sending window. The window mechanism is for TCP flow control which prevents a faster sender from overwhelming a slower receiver. The high system idle rate means the tasks on the system are often blocked, which is caused by the small TCP sending buffer size and the TCP flow control mechanism.

TCP flow control artificially slows down the data transfer in the tests by blocking the sending task on the sender. The receiver governs the amount of data sent by the sender by returning an advised window with every acknowledgement indicating an allowed number of octets that the sender may transmit before receiving further permission. But the advised window size sent back by the receiver does not carry enough information for flow control. The window only controls how much data can be sent rather than how fast the transmission should go. For flow control, the most important information needed at the sender is the packet processing rate at the receiver. The sending rate at the
Table 9. System idle time on the sending side for total time to send 40 Kbytes data (numbers represent the sample mean and, in parentheses, the half length of the confidence interval with 95% confidence level).

<table>
<thead>
<tr>
<th>System Idle Time</th>
<th>Total Time</th>
<th>Percentage of Time the system is idle</th>
</tr>
</thead>
<tbody>
<tr>
<td>msec</td>
<td>msec</td>
<td></td>
</tr>
<tr>
<td>156.491 (8.65)</td>
<td>345.452 (16.00)</td>
<td>45.3%</td>
</tr>
</tbody>
</table>

sender should be decided by the receiver processing rate to achieve high throughput. However, the advised window, the size of which is decided by the buffer size on the receiver, only specifies how much data the sender can transfer before it receives further permission. The sender knows nothing about the receiver processing rate from the advised window. Since the window mechanism does not provide this information to the sender, the data transfer would not occur at an optimal rate. An alternative for the window mechanism could be a rate-based flow control mechanism [10], which provides a way for the receiver to directly control the sending rate. Any improvement of the TCP
flow control mechanism would result in considerable improvement of the performance of TCP-based bulk data transfer.

The high fraction of time during which the system is idle in the tests is also due to the small TCP sending buffer size. Ideally, the user task moves user data to the TCP sending buffer without being blocked. This could happen only if the size of the TCP sending buffer is large enough so that by the time the user task fills the sending buffer, the acknowledgement for the first packet has arrived to free part of the sending buffer. Suppose the average round trip time for a packet is \( t \) seconds and the user task copies the user data to the sending buffer at a rate of \( r \) bytes/second. Then, to achieve the desirable situation, the TCP sending size \( l \) should at least be \( t \times r \) bytes. If retransmission is considered, \( l \) should be even larger.

### 4.2.3 Some Speed Predictions

If the overhead of the execution of TCP/IP protocol were the only bottleneck, how fast could TCP/IP forward data? This section contains some speed predictions of this maximum data transfer rate based on the measurement of TCP/IP protocol execution time.

Tests in this section involve a PS/2 and a Dell/310 system. The instrumented version of TCP/IP is run on D310 while the original TCP/IP is used on a PS2. The tests are carried out by running the sockets version performance testing program from Section 2.2 on PS2 and D310. Each test involves establishing a TCP connection, transmitting 1024 bytes of data using 1024 bytes as the writing size, and closing the TCP connection. The instrumented version of TCP/IP on D310 records the TCP/IP protocol execution time on each test. The testing program is run in two ways, First, the sending program is run on D310 and the receiving program is run on PS2. Then the sending program is run on PS2 and the receiving program is run on D310. By running the sending and the receiving program on D310, results for both the sender and receiver are obtained. Each test
is repeated 100 times. The test results shown in Table 10 are the average over 100 trials. Note the
time in Table 10 is the protocol execution time; it does not include the data copy time.

Assume the receiving TCP acknowledges every other data segment, it takes TCP/IP 851.5 micro-
seconds \((647 + 0.5 \times 409)\) to transmit 1 Kbyte on the sending side and 734.5 microseconds \((599 + 0.5 \times 271)\) to receive 1 Kbyte on the receiving side. So it is justifiable to assume that it is possible
to transfer 1 Kbyte of data within 851.5 microseconds on either the sending side or the receiving
side. The testing machine has an Intel-80386 CPU running at 12.5MHZ. A PC machine with a
33MHZ Intel chip can execute the same TCP/IP instructions in 322.5 microseconds, or 1024
bytes/322.5 microseconds = 3,175 Kbytes/sec. This yields a throughput of 25.4 Mbps.

Since TCP/IP is a general-purpose communication protocol with a lot of functionality, it is often
suspected to be a major source of the network communication overhead. But the tests results in-
dicate the execution of TCP/IP protocol itself is not a serious source of overhead. The numbers
mean that even on a relatively slow PC, the TCP/IP protocol will not become the data transfer
bottleneck on a high speed network. This means it is not necessary to put TCP/IP in hardware as
suggested in [22].

\[ \text{4.3 The Overhead of TCP Segment Processing} \]

Although the last section shows TCP/IP is not the major overhead of packet processing, it is still
useful to identify the overhead of TCP segment processing. This section analyzes the MD-DOS/IP
package TCP implementation and gives some implementation suggestions based on the analysis.
Table 10. Cost for execution of TCP/IP protocol for a 1024 bytes segment (numbers represent the sample mean and, in parentheses, the half length of the confidence interval with 95% confidence level).

<table>
<thead>
<tr>
<th></th>
<th>Sender</th>
<th></th>
<th>Receiver</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sending Data (microsec)</td>
<td>Processing ACK (microsec)</td>
<td>Receiving Data (microsec)</td>
<td>Sending ACK (microsec)</td>
</tr>
<tr>
<td>TCP</td>
<td>523 (0.61)</td>
<td>297 (0.67)</td>
<td>493 (5.20)</td>
<td>153 (0.39)</td>
</tr>
<tr>
<td>IP</td>
<td>124 (0.53)</td>
<td>112 (0.46)</td>
<td>106 (0.32)</td>
<td>118 (1.25)</td>
</tr>
<tr>
<td>Total</td>
<td>647</td>
<td>409</td>
<td>599</td>
<td>271</td>
</tr>
</tbody>
</table>

4.3.1 Test Procedure

The tests involve a PS/2 and a Dell/310 system. The instrumented version of TCP/IP is run on D310 while the original TCP/IP is used on PS2. The tests are carried out by running the sockets version performance testing program from section 2.2 on PS2 and D310. Each test involves establishing a TCP connection, transmitting user data, and closing the TCP connection. The amount of data transferred in the tests are 1, 10, 100, 500, 1024, and 1460 bytes. The writing size is chosen
to be the same as the total amount of transferred data. Thus, the TCP connection is closed after each call to \textit{write}. By doing so, the size of the TCP data segment in all the tests is made equal to the writing size. The test is repeated 100 times for each writing size. The instrumented TCP/IP records the time each packet spends in different TCP parts. The tests results obtained are the average over 100 trials.

4.3.2 Test Results and Observations

4.3.2.1 Sending Side TCP

The results in this section are obtained by running the sending program on D310 and the receiving program on PS2. Figure 14 shows the breakdown of the TCP segment processing time. The data size is the number of octets carried by a data segment. Note that the segment processing time also includes the processing of the acknowledgement for the segment. As expected, the cost for memory copy and checksum increases as the data size increases. The cost for the other parts is constant across different data sizes.

Timer management includes timer setting and acknowledgement processing. The sending side TCP starts a timer each time a segment is sent. On receiving an acknowledgement, TCP calculates the round trip time. Based on the new round trip time, TCP updates the retransmission time out value and sets the sending window size. The MD-DOS/IP TCP uses Jacobson's round trip timing and slow start algorithms [14] to set these two values. The cost for the timer management is 86 microseconds for each data segment. Out of 86 microseconds, 22 microseconds is for the timer setting when the segment is sent out, 64 microseconds is for the acknowledgement processing. Even though the relatively complicated Jacobson's algorithms are used, the cost for the timer management is low.
Figure 14. Overhead of TCP segment processing on the sending side
The data copy and checksum computation become the major overhead of TCP when the data size is large. In fact, when the data size is 1460 bytes, TCP spends 36% of its total segment processing time on the data copy and 33% on the checksum computation.

It is questionable whether the end-to-end TCP checksum is necessary since the data link layer has already checksummed every data frame. Unlike the data link layer checksum, which is normally done by hardware, the TCP checksum is computed by the central processor. It is clear that the redundant checksum has performance penalties [9].

Clark, et al. [17] suggest a performance optimization by noticing that the checksum of the data can be computed in the same loop as the data copy. In this TCP implementation, routine \texttt{tcp\_write()} is for data copy and routine \texttt{tcp\_cksum()} computes the checksum. These two routines can be combined to use a single loop for both data copy and checksum computation.

In some applications, such as transmitting disk files over a network, the user process first copies the data into its address space (e.g., from disk), then it calls TCP which moves the data from the user address space to the TCP sending buffer. The TCP data copy can be avoided in this kind of applications if the TCP/user interface were redesigned. In the new design, the TCP sending buffer is made visible to the user. When the user wishes to send data, it applies to TCP for the sending buffer. After getting the buffer, the user copies the data directly to the TCP sending buffer bypassing the user memory. So instead of doing two copies, only one data copy is needed. TCP checks the sending buffer periodically for the user data it could send. One problem with this scheme is that it makes TCP vulnerable to a user programming error. It is also possible that the user process may hold the sending buffer without using it.
4.3.2.2 Receiving Side TCP

The results in this section are obtained by running the sending program on PS2 and the receiving program on the D310. Figure 15 shows the breakdown of the TCP segment processing time on the receiving side. The segment processing time also includes the acknowledgement preparation time for the segment.

Unlike the sending side TCP, there is no timer management on the receiving side. The data copy and checksum computation also become the major overhead when the data size is large. The situation here is very similar to that on the sending side.

It is also possible to put the data copy and checksum computation in the same loop on the receiving side as suggested in [17]. But on the receiving side, the effect of the data copy must be ignored if the data is found to be corrupted. This requirement brings more complexity to TCP. In this TCP implementation, routine tcp_timers() is responsible for copying the data from the TCP assembly line to the user buffer and routine tcp_cksum() computes the TCP checksum. These two routines can be combined to use a single loop to do the data copy and checksum. To counter the effect of the data copy if the data is found to be corrupted, the user buffer keeps a current address pointer indicating where the TCP can copy the next data. The user buffer current address pointer changes as TCP copies the data to the user buffer. Before TCP begins to copy data from a data segment to the user buffer, it records the user buffer current address pointer. If the TCP checksum shows the data segment is corrupted, the TCP sets the user buffer current address pointer to the pointer it recorded before the data copy. In this way, corrupted data would not affect the normal data copy.

On the receiving side, data is first moved off the network board to the TCP/IP buffer and then copied to the user buffer from the TCP/IP buffer. The later data copy can be avoided if instead of copying the data to the user buffer, TCP passes a buffer pointer to the user process. The buffer pointer points to the user data in the TCP/IP buffer. For those applications which do not care
Figure 15. Overhead of TCP segment processing on the receiving side
about the location of the received data as long as the data is accessible from the user process, this scheme can save one data copy. To support all applications, TCP can be redesigned to provide both pointer passing and direct data copy methods to the applications.

4.4 The Bottleneck of Data Communication Using TCP/IP

Section 4.2 shows that, while the execution of the TCP/IP protocol is not the major overhead of the packet processing, TCP's window mechanism contributes to the high system idle rate, and hence reduces the user-perceived throughput. There are four factors that effect the user-perceived TCP throughput:

- the transmission medium rate,
- the packet sending rate at the sending side (termed "sending rate"),
- the packet processing rate at the receiving side (termed "receiving rate"), and
- the receiving side buffer size.

In this research, the transmission medium rate is not considered because the transmission medium is not the data transfer bottleneck in this study. Section 4.4.1 analyzes the remaining three factors based on queueing theory. Section 4.4.2 gives the test results which confirm the analysis results.
4.4.1 Analysis

Consider a bulk data communication which has no flow control. The sender transfers data to the receiver at its highest sending rate. At the receiver, arriving packets are stored in the receiving buffer and then processed by the receiver. The receiver can accommodate at most \( k \) packets. The receiver can be modeled as an \( M/M/1/k \) queue. The reason for using \( M/M/1/k \) queue is that in this research, each test involves transferring 40 Kbytes of data and no other user process is competing for the CPU time, so the transient case is avoided. For convenience, we assume the sending and receiving process are Markov process. We believe \( M/M/1/k \) queueing model gives correct qualitative description of the receiver. Let the average receiving rate be \( \mu \) packets/sec. Let the average sending rate be \( \lambda \) packets/sec. Since the transmission medium is not the bottleneck, the average arrival rate at the receiver is the average sending rate. Traffic intensity \( \rho \) is defined as \( \lambda/\mu \).

If \( \rho < 1 \) (the average sending rate is lower than the average receiving rate), according to queueing theory, the probability that packets are turned away and not accepted by the receiver because the buffer is full, \( P_{\text{no, buffer}} \), is \( (1 - \rho) \times \rho^{k}/(1 - \rho^{k+1}) \) [23, p. 36]. With \( \rho < 1 \) and \( k > 1 \), we have \( P_{\text{no, buffer}} \approx (1 - \rho) \times \rho^{k} \). Figure 16 shows the buffer size as a function of \( \rho \).

Figure 16 shows that when using a buffer size of 6 packets, the probability that the receiving buffer overflows is only 0.01 if \( \rho \) is less than 0.5. In another word, when \( \rho \) is small (e.g., less than 0.5), the sender can send data up to the maximum rate of \( \lambda \) without overwhelming the receiver even if the receiving buffer size is relatively small. This is because when \( \rho \) is small, the sending side is the data transfer bottleneck. This result is also true for TCP because the major difference between our model and TCP is that TCP has a flow control mechanism which only slows down the sending rate.

When the average receiving rate is close to the average sending rate (\( \rho \approx 1 \)), one might naïvely believe that the sender should be allowed to transmit data as fast as possible to achieve high throughput. This is not true because of the randomness of the packet arrival and processing processes. Even
Figure 16. The receiving buffer size as a function of $\rho$
though the average receiving rate and average sending rate is close, it is possible that packets arriving at the receiver may not get processed immediately because the receiver is busy with other activities. Thus, many receiving buffers are needed to hold incoming packets when the receiver experiences a long service time. In fact, the size of the receiving buffer increases rapidly as $\rho$ increases (see Figure 16). Because the receiving buffer size is limited, the receiver must throttle the sending process at some point to avoid buffer overflow at the receiver. In TCP, a window mechanism is used to control the sending rate. The receiving TCP controls the sending rate by sending back an advised window in every acknowledgement indicating an allowed number of octets that the sender can send before receiving another acknowledgement. TCP implementations generally set the advised window to the size of the available receiving buffer. A smaller receiving buffer means the sending TCP will be stopped more often, hence reducing the throughput. The receiving TCP should choose the size of the receiving buffer as large as possible to increase the throughput.

When the average sending rate is much higher than the average receiving rate, flow control must be used to ensure the sender does not overwhelm the receiver. As long as the receiving process can always get data from the receiving buffer, the user-perceived TCP throughput will be equal to the receiving rate. Therefore, if the receiving buffer is large enough to guarantee the receiving process does not run out of data in its receiving buffer, the TCP data transfer bottleneck is the receiver. When the receiver becomes the data transfer bottleneck, using a larger receiving buffer does not help much to improve the TCP throughput.

4.4.2 Tests Results and Observations

The purpose of the tests in this section is to verify the bottleneck analysis results. That is, we want to look at throughput as a function of various relative sending/receiving rates (e.g., various values for $\rho$). The tests involve a Dell310, a PS2, a Macintosh II, and a DECstation 5000/200 (machines slower, equal, and faster than D310). The sockets version TCP performance testing program is run
between three machine pairs, MacII to D310, PS2 to D310, and DEC5000 to D310. The sending program on the MacII is purposely slowed down by inserting some extra operations between each write to get a sending rate lower than the D310 receiving rate. The three machine pairs correspond to three situations we are interested in, namely, the sending rate much lower than, close to, or much higher than the receiving rate ($\rho < 1$, $\rho \approx 1$, $\rho > 1$). In all the tests, the D310 is the receiver and the writing size is 1 Kbyte. Each test transmits 40 Kbytes of data and is repeated 100 times. The results are averaged over 100 trials.

The maximum receiving rate on D310 is obtained by directly measuring the cost associated with running TCP/IP on D310. The measurement shows that it takes the receiver 3.469 milliseconds to process a 1024 bytes data segment (this includes the processing time of the TCP, IP, and network driver) and 0.477 milliseconds to prepare an acknowledgement. Since MD-DOS/IP TCP acknowledges at least every other packet in bulk data transfer, the cost to process and acknowledge a 1024 byte packet is at least $3.469 + 0.5 \times 0.477 = 3.71$ milliseconds. So the upper bound for the maximum receiving rate $= \frac{1024 \text{ bytes}}{3.71 \text{ milliseconds}} = 2.209 \text{ Mbps}$.

The maximum sending rate for each TCP sending program on a certain host is estimated by measuring the rate at which the host can transmit UDP datagrams to a faster receiver. For example, the maximum sending rate for the MacII TCP sending program is estimated by running the UDP testing program between the MacII sender and a DEC5000 receiver. Since UDP has no flow control and transmits data as fast as possible, the UDP sending rate is equal to its throughput. The MacII UDP sending rate is used as an approximation to the MacII TCP maximum sending rate. The maximum sending rate for the TCP sending program on the PS2 is estimated in the same way. The maximum sending rate for DEC5000 is estimated by measuring the UDP throughput between two DEC5000s.

Figure 17 shows the TCP data transfer throughput on different machine pairs. The receiving rate represents the maximum D310 receiving rate (2.209 Mbps). The sending rate is a curve drawn through the maximum sending rate for the TCP sending program on MacII, PS2, and DEC5000.
which is 0.552 Mbps, 2.596 Mbps, and 8.192 Mbps, respectively. The remaining lines represent the
actual TCP throughput on different machine pairs under varying TCP receiving buffer sizes.

\( \rho \ll 1 \). For the MacII and D310 machine pair, Figure 17 shows that the throughput does not
change much for different receiving buffer sizes. This is because the ratio of the sending rate to the
receiving rate is 1:4, and according to the bottleneck analysis in last section, the sending side is the
bottleneck of data transfer when \( \rho \) is small. Therefore, the throughput is limited by the sending rate,
not the window size (based on the receiving buffer size).

\( \rho \approx 1 \). For the PS2 and D310 machine pair, the ratio of the sending rate to the receiving rate is
1.17:1. In this case \( \rho \) is close to 1, the receiving buffer becomes the data transfer bottleneck. So in-
creasing the receiving buffer size helps to increase the throughput as indicated by Figure 17.

\( \rho > 1 \). For the DEC5000 and D310 machine pair, the ratio of the sending rate to the receiving rate
is 3.8:1. Figure 17 shows the throughput increases from 1.677 Mbps to 2.06 Mbps as the receiving
buffer size increasing from 1K to 2K. This is because when the receiving buffer is 1K, the receiving
buffer can at most hold one packet (the average packet size is 1K). After processing the packet in
the receiving buffer, the receiver must wait for another packet. When using 2K as the receiving
buffer size, the receiving buffer can hold 2 packets. After processing a packet, the receiver can keep
on processing another packet and, at the same time, receive a new packet from the sender. Hence,
the receiver can avoid the wait and increase the throughput. It is noted that when the receiving
buffer size is greater than 2K, the throughput does not change much under different receiving buffer
sizes. This is because as long as the receiver can always find a packet in its receiving buffer, the
throughput is decided by the receiving rate.

The D310 is a slow machine by today's standards, so we redid tests with a RISC architecture
workstation. Three cases are studied in the new tests, namely, the sending rate is much lower than,
very close to, and much higher than the receiving rate. The sockets version TCP performance
testing program is used in the tests to measure the TCP throughput with the receiving program
Figure 17. The TCP throughput for different machine pairs (Dell310 as the receiver)
running on a DEC2100. To study the three cases we are interested in, the performance testing program is run on three machine pairs, a MacII to the DEC2100, a DEC5000 to the DEC2100 with the sending program on the DEC5000 being purposely slowed down, and a DEC5000 to the DEC2100. The maximum receiving rate for the DEC2100 is estimated by measuring the highest rate at which the DEC2100 can accept UDP datagrams from a fast machine without losing data, which is 4.2 Mbps. The maximum sending rate for the TCP sending program on different machines is estimated using the UDP version performance testing program as discussed earlier. Figure 18 shows the results. The receiving rate represents the DEC2100 receiving rate. The sending rate is a curve drawn through the sending rate of the TCP sending program on MacII, “slow” DEC5000, and DEC5000, which is 2.074 Mbps, 3.9 Mbps, and 9.04 Mbps, respectively. The actual TCP throughput on different machine pairs is shown by the other curve.

\( \rho < 1 \). For the MacII to the DEC2100, the ratio of the sending rate to the receiving rate is 1:2.03. When the receiving rate is much higher than the sending rate, according to the bottleneck analysis, the bottleneck is the sending side. This is substantiated by Figure 18 which shows that the TCP throughput for the MacII to DEC2100 case is close to the sending rate.

\( \rho \approx 1 \). For the “slow” DEC5000 to the DEC2100, the ratio of the sending rate to the receiving rate is 1:1.07. Figure 18 shows the TCP throughput for the “slow” DEC5000 to DEC2100 case is far below the sending rate and the receiving rate because when the sending rate is close to the receiving rate, the receiving buffer is the data transfer bottleneck.

\( \rho > 1 \). For the DEC5000 to DEC2100 case, the ratio of the sending rate to the receiving rate is 2.15:1. When the sending rate is much higher than the receiving rate, the receiving side becomes the data transfer bottleneck. This is confirmed by Figure 18, which shows the TCP throughput for the DEC5000 to the DEC2100 is close to the receiving rate.

To summarize, when the sending rate is far less than the receiving rate (\( \rho < 0.5 \)), the sending side is the data transfer bottleneck. By using a large enough receiving buffer size (e.g., large enough to
Figure 18. The TCP throughput for different machine pairs (DEC2100 as the receiver)
hold 6 packets), the sender can transmit data as fast as possible without overwhelming the receiver ($P_{no\_buffer} = 0.01$). The receiving buffer becomes the data transfer bottleneck when the sending rate and the receiving rate is close ($\rho$ near 1). So the receiving buffer should be made as large as possible to increase the throughput. When the sending rate is much higher than the receiving rate ($\rho > 1$), the receiving side becomes the data transfer bottleneck. As long as the receiver can always find a packet in its receiving buffer, the throughput does not change much under different receiving buffer sizes.

4.5 Summary

This chapter investigated the overhead of TCP/IP packet processing and TCP segment processing. The test environment was described. In the first part, the overhead of TCP/IP packet processing was identified. Based on the study results, the problem with the TCP flow control mechanism was discussed. Also discussed was the TCP sending buffer size. Some TCP/IP speed predictions were made. In the second part, the overhead for TCP segment processing was identified. The data copy and checksum are the two major overheads of TCP segment processing. Some TCP implementation suggestions were given based on the overhead study. Finally, the bottleneck of data communication using TCP/IP was analyzed.
Chapter 5. Summary and Conclusions

This research investigated end-to-end flow control and its effect on the user process communication performance in a local area computer network. Results in this research are based on experimental measurements and analysis of these measurements. The research were conducted in two parts.

In the first part, the effects that the following factors have on user-perceived delay and throughput were examined for two transport layer protocols, namely, TCP and UDP:

- different processor types,
- load on the host machine, and
- choice of application program interface to the communication protocol.

In the second part, the overhead of TCP/IP packet processing and TCP segment processing was identified based on the TCP/IP implementation of the MD-DOS/IP package. Some speed predictions for the TCP/IP protocol were made if all costs besides TCP/IP were made zero. This part also gave some TCP implementation suggestions. Furthermore, the bottleneck of data communication using TCP/IP was studied.
5.1 Effects of Different Processors

The following conclusions on the impact of different speed processors on communication performance can be drawn from the measurements in Chapter 3.

1. The speed of host processor has a profound impact on the user-perceived communication performance. Generally, the RISC (SUN and DECstation) and 68040-based (NeXTstation) hosts cluster in one group and the 68020, 68030, and 80386 hosts in a slower cluster. The first cluster has logarithmic growth of throughput with write size for a given amount of data to transmit, while the slower cluster is linear. For UDP, the first cluster outperforms the slower cluster by a factor of at least 2 to 3 times for all writing sizes.

2. The one post-RISC CISC host (NeXT) did almost as well as RISC (SUN) in terms of throughput.

3. For a small writing size, TCP yields better throughput than UDP since it batches user data to reduce the packet processing overhead. But as the writing size increases, UDP offers better performance than TCP because TCP's flow control mechanism slows down the user data flow.

4. TCP employs a window mechanism to control the data flow. The receiver governs the sending rate by sending back an advised window with every acknowledgement. The advised window indicates an allowed number of octets that the sender may transmit before receiving further permission. The size of the available receiving buffer at the receiver determines the advised window size. So using a larger receiving buffer on the receiver (e.g., as in the DECstation) increases the sending rate, resulting in higher user-perceived throughput on TCP, regardless of the speed of any network component (sending CPU, receiving CPU, network adaptor, communication media).
5. Our measurements confirms that when using UDP, it is essential to prevent a faster sender from overwhelming a slower receiver with data (e.g., as for the PS2 to Dell310). Since UDP has no flow control, the user is responsible for balancing the sending rate and the receiving rate.

5.2 Effects of Host Load

From the study of the effect of host load on the communication program, the following can be concluded.

1. When few packets are generated by the network software, delay and throughput generally varied little with host load. Therefore, protocol software overhead is not the dominate performance problem for networks under this situation.

2. User-perceived network delay becomes more sensitive to the host load as the amount of data packets generated by the network software increases. To reduce the influence of host load on communication programs, it is helpful to minimize the number of packets generated by the underlying network software for the user program. One strategy is to use a larger writing size in the user process and, if using TCP, another strategy is to increase the segment flush timeout value. Both strategies help decrease the number of packets generated by the underlying network software and hence reduce the influence of host load on the performance of the communication program.
5.3 Effects of Application Program Interface

From the study of the effects of the application program interface on the communication program, the following can be concluded.

The application program interface does affect the performance of the communication program. In the tests, a delay disparity as large as 24% is observed when sockets and TLI are used for the same communication program. Our test results show that sockets achieves better performance than TLI when using TCP, but TLI outperforms sockets when UDP is used. The test results indicate it is as important to optimize the performance of the application program interface as it is to optimize the protocol implementation.

5.4 The Overhead of TCP/IP Packet Processing

The following conclusions are drawn from the study of the overhead of TCP/IP packet processing.

1. TCP flow control often introduces a major inefficiency to the packet processing in that it forces the sender, receiver, and communication medium to be artificially idle. The defects of a window mechanism for flow control worsen the situation. The problem with the window mechanism arises because the sender does not know the most important flow control information, which is the packet processing rate at the receiving side. So the data transfer does not occur at an optimal rate. This suggests that a better flow control mechanism, such as rate-based flow control, should be used for protocols designed for high performance.
2. To reduce the system idle time at the sender, the TCP sending buffer should be large enough so that the user process does not block because the sending buffer is full. Specifically, the sending buffer size \( l \) should be greater than \( t \times r \), where \( t \) is the average packet round trip time and \( r \) is the rate at which the user process copies the data to the sending buffer.

3. Although the TCP/IP protocol suite offers a lot of functionality, the protocol implementations themselves are not a serious source of overhead. Based on our test results, we predicted that when running on high-speed PC's, TCP/IP can achieve a throughput of 25.4 Mbps on a local area network. This means it is not necessary to put TCP/IP in hardware for most applications and networks.

5.5 The Overhead of TCP Segment Processing

From the study of the overhead of TCP segment processing, the following conclusions are drawn.

1. The data copy is a major overhead of TCP segment processing. When the data size is 1460 bytes, TCP spends over 30% of its total processing time on the data copy. By redesigning the user-to-TCP interface as suggested in Section 4.3, some data copy time can be saved.

2. The redundant TCP checksum has heavy performance penalties. The TCP checksum becomes one of the major overheads of TCP segment processing as shown in Section 4.3. One performance optimization [17] is to have a single loop to do the data copy and the checksum computation. Section 4.3 also discusses how to apply this optimization to the MD-DOS/IP package.

3. The cost for TCP timer management is low even though relatively complicated algorithms are used.
5.6 The Bottleneck of Data Communication Using TCP/IP

Based on the bottleneck study of data communication using TCP/IP, the following conclusions are drawn.

1. When the sending rate is far less than the receiving rate ($\rho < 0.5$, where $\rho$ is defined as the ratio of the average sending rate to the average receiving rate), the sending side is the data transfer bottleneck. Even with only a relatively small receiving buffer, the sender can transmit data as fast as possible without overwhelming the receiver.

2. When the sending rate is close to the receiving rate ($\rho \approx 1$), the receiving buffer is the data transfer bottleneck. Choosing a receiving buffer size as large as possible helps to improve the throughput.

3. When the sending rate is much higher than the receiving rate ($\rho >> 1$), the receiving side becomes the data transfer bottleneck. As long as the receiver can always find a packet in its receiving buffer, the throughput does not change much under different receiving buffer sizes.
Bibliography


[20] Computer Science Center, University of Maryland, MD-DOS/IP Installation Guide. Package Number 322.


Appendix A. TCP with Sockets

/*-----------------------------*/
/* TCP Socket: sending program */
/*-----------------------------*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <sys/ioctl.h>

#define BUFSIZE 4096
#define sdomain AF_INET
#define sotype SOCK_STREAM
#define SERV_TCP_PORT 5010 /* define port #
 must greater than 5000 */

main()
{
 int so, nso;
 struct sockaddr_in sockname,raddr;
 int namelen = sizeof(sockname),rsize = sizeof(raddr);
 int i,j,k,dd,remainder,grainsize,replica;
 char c,msg[BUFSIZE];
 FILE *fp;
 unsigned long size;

 /*-----------------------------read in control parameters---------------*/
 fp = fopen("para","r");
 fscanf(fp,"%d %lu %d",&replica,&size,&grainsize);
 fclose(fp);

 /*-----------------------------Now, carry out the experiments--------------*/
 for(k = 0;k < replica;k + + )
 {
  /* Create a socket */
  so = socket(AF_INET,SOCK_STREAM,0);
  if(so == -1) perror("socket"); /* Return Error */

  /* Bind the socket to a name */
  /* Prepare the name */
  bzero((char *)&sockname,sizeof(sockname));
}
sockname.sin_family = AF_INET;
sockname.sin_port = htons(SERV_TCP_PORT + k);
sockname.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(so,(struct sockaddr *)&sockname,sizeof(sockname)) < 0)
    perror("bind");

/* Listen to a socket */
if(listen(so,5) < 0) perror("listen");

/* Accept a connection on the socket */
nso = accept(so,(struct sockaddr *)&addr,&rsize);
if(nso == -1) perror("accept");

/*-----------------send the data on socket------------------*/
dd = size / grainsize; remainder = size % grainsize;
for(i = 0;i < grainsize;i++) msg[i] = 'T';

for(j = 1;j < dd;j++)
{
    if(write(nso,msg,grainsize) != grainsize)
        { perror("write error"); exit(0);}
}

/*------------------the last packet contains char 'e'------------------*/
if(remainder == 0) msg[0] = 'e';
if(write(nso,msg,grainsize) != grainsize)
    { perror("write error"); exit(0);}

msg[0] = 'e';
if(remainder != 0)
    if(write(nso,msg,remainder) != remainder)
        { perror("write error"); exit(0);}

while(read(nso,&c,1) != 1);

close(nso);
close(so);

}


/*---------------------------------------------*/
/* TCP socket: receiving program */
/*---------------------------------------------*/

#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/tcp.h>
#include <sys/time.h>
#include <time.h>
#include <math.h>
#define BUFFSIZE 4096
#define sodomain AF_INET
#define sotype SOCK_STREAM
#define SERV_TCP_PORT 5500 /* define the receiving port */
#define SERV_TCP_HOST "128.173.6.161" /* larry Internet add.
 can be found in /etc/hosts */
#define REPORT "tcp"

main()
{
int so, nso,fd;
struct sockaddr_in sockname,radd;
int namelen = sizeof(sockname),rsize = sizeof(radd);
int i,j,k,remainder,grainsize,replica;
char c[msg[BUFFSIZE],c][10];
FILE *fp,*ip1;
struct timeval btime_buff, etime_buff;
double stop,start,avgthr = 0,sdev = 0,thr,dtime;
unsigned long size,total = 0,elptime;
time_t tm;

/*-------------------read in control parameters--------------------------*/
fp = fopen(\"para\",\"r\");
scanf(fp,\"%d %lu %d\",&replica,&size,&grainsize);
fclose(fp);

/*---------Now, carry out the experiments-----------------------------*/
for(k = 0;k < replica;k + +)
{
  /* Create a socket */
  so = socket(AF_INET,SOCK_STREAM,0);
  if(so = = -1) perror(\"socket\"); /* Return Error */

  /* Prepare the name */
  bzero((char *)&sockname,sizeof(sockname));
  sockname.sin_family = AF_INET;
  sockname.sin_port = htons(SERV_TCP_PORT + k);
  sockname.sin_addr.s_addr = inet_addr(SERV_TCP_HOST);

  /* Connect to a port on another machine */
  nso = connect(so,(struct sockaddr *)&sockname,sizeof(sockname));
  if(nso = = 0) perror(\"connect\");

  /*----------Receive data from socket-------------------------------*/
  printf(\"before receive data\n\");
  total = 0;
  i = read(so,msg.grainsize);
  if(i > 0) total + = i;

  gettimeofday(&btime_buff);
  while(total = size)
  {
    i = read(so,msg.grainsize);
  }

Appendix A. TCP with Sockets
if(i > 0) total += i;
};
gmtimeofday(&etime_buff);

c = 'e';
while(write(so,&c,1)!=1);

close(fd);
close(so);

 /**************************************************************************/
/** Compute the delay ****************************************************/
start = ((double)bt ime_buff.tv_sec) *1000 + bt ime_buff.tv_usec/1000;
stop = ((double)etime_buff.tv_sec) *1000 + etime_buff.tv_usec/1000;
elptime = stop - start;

fp = fopen(REPORT,"a");
fprintf(fp,"%lu\n", elptime);
fclose(fp);

printf("End of Client\n");
sleep(2);
}
Appendix B. UDP with Sockets

/*---------------------------------------------*/
/* UDP socket: sending program */
/*---------------------------------------------*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/udp.h>
#include <sys/ioctl.h>

#define BUFFSIZE 4096
#define sdomain PF_INET
#define soctype SOCK_DGRAM
#define SERV_UDP_PORT 6010 /* define port #
                         must greater than 5000 */

main()
{
    int so, nso;
    struct sockaddr_in sadd, radd;
    int namelen = sizeof(sadd), rsize = sizeof(radd);
    int i,j,k,dd,remainder,grainsize,replica;
    char c, msg[BUFFSIZE];
    FILE *fp;
    unsigned long size;

    /*-------------------read in control parameters-------------------*/
    fp = fopen("para","r");
    fscanf(fp,"%d %lu %d ",&replica,&size,&grainsize);
    fclose(fp);

    /*----------------Now, carry out the experiments----------------*/
    for(k = 0; k < replica;k + + )
    {
        /* Create a socket */
        so = socket(PF_INET,SOCK_DGRAM,0);
        if(so == -1) perror("socket"); /* Return Error */

        /* Bind the socket to a name */
        /* Prepare the name */
        bzero((char *)&sadd,sizeof(sadd));
sadd.sin_family = PF_INET;
sadd.sin_port = htons(SERV_UDP_PORT + k);
sadd.sin_addr.s_addr = htonl(INADDR_ANY);

if(bind(so,(struct sockaddr *)&sadd,sizeof(sadd))<0)
  perror("can't bind local address");

/* Accept data request on the socket */
if(recvfrom(so,msgr.grainsize,0,(struct sockaddr *)&radd,
            &rsze)!=grainsize)
  perror("data request failure");

/*********send the data on socket***********/
dd = size / grainsize; remainder = size % grainsize;
for(i = 0;i < grainsize;i ++) msg[i] = '1';

for(j = 1;j < dd;j++)
{
  if(sendto(so,msgr.grainsize,0,(struct sockaddr *)&radd,
            &rsze)!=grainsize)
    { perror("write error"); exit(0); }
}

/*--------the last packet contains char 'e'--------*/
if(remainder = = 0) msg[0] = 'e';
  if(sendto(so,msgr.grainsize,0,(struct sockaddr *)&radd,
            &rsze)!=grainsize)
    { perror("write error"); exit(0); }

if(remainder!=0)
  if(sendto(so,msgr,remainder,0,(struct sockaddr *)&radd,
            &rsze)=remainder)
    { perror("write error"); exit(0); }

close(so);

printf("End of Server
");
}


/*-------------------------------*/
/* UDP socket: receiving program */
/*-------------------------------*/
#include <stdio.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <netinet/udp.h>
#include <sys/time.h>
#include <time.h>

#define BUFSIZE 4096
#define sdomain AF_INET
#define sotype SOCK_DGRAM
#define SERV_UDP_PORT 5500 /* define the receiving port #
must greater than 5000 */
#define SERV_UDP_HOST "128.173.6.161" /* larry Internet add.
can be found in /etc/hosts */
#define REPORT "udp"

main()
{
  int so, nso,fd;
  struct sockaddr_in saddr,raddr,rv;
  int namelen = sizeof(saddr), rsize = sizeof(raddr);
  int i,j,k,remainder,grainsize,replica;
  char c, msg[BUFSIZE];
  FILE *fp,*fp1;
  struct timeval btime_buff,etime_buff;
  double stop,start,avgthr = 0,sdev = 0,avgerr = 0,thr,dtime;
  unsigned long size,total = 0,elptime;
  time_t tm;

  /*------------------read in control parameters-----------------*/
  fp = fopen("para","r");
  fscanf(fp,"%d %lu %d",&replica,&size,&grainsize);
  fclose(fp);

  /*--------Now, carry out the experiments------------------------*/
  for(k = 0;k < replica;k++)
  {
    /* Prepare the server name */
    bzero((char *)&saddr,sizeof(saddr));
    saddr.sin_family = AF_INET;
    saddr.sin_port = htons(SERV_UDP_PORT + k);
    saddr.sin_addr.s_addr = inet_addr(SERV_UDP_HOST);

    /* Create a socket */
    so = socket(AF_INET,SOCK_DGRAM,0);
    if(so == -1) perror("socket"); /* Return Error */

    /* bind any local address for us */
    bzero((char *)&raddr,sizeof(raddr));
    raddr.sin_family = AF_INET;
    raddr.sin_port = htons(0);
    raddr.sin_addr.s_addr = htonl(INADDR_ANY);

    if(bind(so,(struct sockaddr *)&raddr,sizeof(raddr)) < 0)
      perror("client: can't bind local address");

    /*----------send a data request to server-------------*/
    if(sendto(so, msg,grainsize,0,(struct sockaddr *)&sadd,
      sizeof(saddr)) != grainsize)
      perror("data request error");

    /*----------Receive data from socket------------------*/
    total = 0;
}
i = recvfrom(so, msg, grainsize, 0, (struct sockaddr *)0, (int *) &namelen);
if(i > 0) total += i;

gettimeofday(&btime_buff);

while( msg[0] != 'e') {
    i = recvfrom(so, msg, grainsize, 0, (struct sockaddr *)0, (int *) &namelen);
    if(i > 0) { total += i; }
}
gettimeofday(&etime_buff);

close(so);

/*-----------------Compute the delay-------------------*/
start = ((double)btime_buff.tv_sec)*1000 + btime_buff.tv_usec/1000;
stop = ((double)etime_buff.tv_sec)*1000 + etime_buff.tv_usec/1000;
elptime = stop - start;

fp = fopen(REPORT,"a");
fprintf(fp,"%lu %lu\n", elptime, total);
fclose(fp);
Appendix C. TCP with TLI

/*---------------------------------------------------------------*/
/* TCP TLI: sending program                                      */
/*---------------------------------------------------------------*/

#include <stdio.h>
#include <fcntl.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stropts.h>

#define DEV_UDP  "/dev/udp"
#define DEV_TCP  "/dev/tcp"
#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 6100
#define MAXSIZE   4096
#define CNTFILE "para"

main()
{
  int tfd;
  struct sockaddr_in cli_addr,serv_addr;
  struct t_bind *req;
  struct t_call *callptr;
  extern int t_errno;
  char msg[MAXSIZE];
  int i,j,k,dd,remainder,grainsize,replica,flag;
  FILE *fp;
  unsigned long size;

  /*-------------------read in control parameters------------------*/
  fp = fopen(CNTFILE,"r");
  fscanf(fp,"%d %lu %d",&replica,&size,&grainsize);
  fclose(fp);

  /*----------------Now, carry out the experiments----------------*/
  for(k = 0;k < replica;k ++ )
  {

Appendix C. TCP with TLI
if((tfd = t_open(DEV_TCP, O_RDWR, (struct t_info *) 0)) < 0)
    perror("server: can't t_open");

/*-----Create a TCP transport endpoint -------*/
bzero((char *) &serv_addr.sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_TCP_PORT + k);

req.addr.maxlen = sizeof(serv_addr);
req.addr.len = sizeof(serv_addr);
req.addr.buf = (char *)&serv_addr;
req.qlen = 2;

if(t_bind(tfd, &req, (struct t_bind *) 0) < 0)
    perror("server: can't bind local address");

/*-----Allocate a t_call stru. for t_listen() and t_accept()-----*/
if((callp = (struct t_call *) t_alloc(tfd, T_CALL, T_ADDR)) == NULL)
    perror("server: t_alloc error for T_CALL");

if(t_listen(tfd, callp) < 0)
    perror("server: t_listen error");

if(t_accept(tfd, callp) < 0)
    if(t_errno == TLOOK){
        if(t_revdis(tfd, (struct t_discon *) 0) < 0)
            perror("t_revdis error");
        if(t_close(tfd) < 0)
            perror("t_close error");
    }

    perror("t_accept error");

/*-----Now, sending data------------------------*/
    dd = size / grainsize; remainder = size % grainsize;
    for(i = 0; i < MAXSIZE; i++) msg[i] = 'T';

    for(j = 1; j < = dd;j++)
        { if(t_snd(tfd, msg, grainsize, 0)! = grainsize)
            { perror("write error"); exit(0); }
        }

    if(remainder! = 0)
        if(t_snd(tfd, msg, remainder, 0)! = remainder)
            { perror("write error"); exit(0); }

    sleep(2);
    close(tfd);
}

/*-------------------------------*/
# include < stdio.h >
# include < fcntl.h >
# include < tiuser.h >
# include < sys/types.h >
# include < sys/socket.h >
# include < netinet/in.h >
# include < stropts.h >
# include < sys/time.h >
# include < time.h >

#define DEV_UDP " / dev/udp "
#define DEV_TCP " / dev/tcp"

#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 6000

#define SERV_HOST_ADDR " 128.173.6.161 " /* loki address*/
#define MAXSIZE 4096
#define REPORT " tlitcp"
#define CNTFILE " para"

main()
{
    int tfd,i,flag,n,k,j,remainder,replica,grainsize;
    struct sockaddr_in cli_addr,serv_addr;
    struct t_call *callptr;
    char msg[MAXSIZE];
    extern int t_errno;
    struct timeval btime_buff,etime_buff;
    double stop,start,avghr = 0,sdev = 0,thr,dtime;
    unsigned long size,total,elptime;
    FILE *fp,*fp1;
    time_t tm;

    /*-------------------read in control parameters-----------------*/
    fp = fopen(CNTFILE,"r");
    fscanf(fp,"%d %lu %d",&replica,&size,&grainsize);
    fclose(fp);

    /*-----------Create the report file-----------------------------*/
    fp = fopen(REPORT,"w");
    fclose(fp);

    /*----------Now ,carry out the experiments---------------------*/
    for(k = 0;k < replica;k + +)
    {
        if((tfd = t_open(DEV_TCP,O_RDWR,(struct t_info *) 0)) < 0)
            perror("client: can't t_open");

        if(t_bind(tfd,(struct t_bind *) 0,(struct t_bind *)0) < 0)
            perror("client: can't bind local address");

        /*-------Create a TCP transport endpoint ---------*/
        bzero((char *) &serv_addr,sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
serv_addr.sin_port = htons(SERV_TCP_PORT + k);

/*----Allocate a t_call stru. for t_listen() and t_accept()----*/
if((callptr = (struct t_call *) t_alloc(ffd,T_CALL,T_ADDR)) == NULL)
    perror("client: t_alloc error for T_CALL");
callptr->addr.maxlen = sizeof(serv_addr);
callptr->addr.len = sizeof(serv_addr);
callptr->addr.buf = (char *) &serv_addr;
callptr->opt.len = 0;
callptr->udata.len = 0;

/*---connect to the server----------*/
n = t_connect(ffd,callptr,(struct t_call *) 0);
if (n < 0) perror("client: can't t_connect to server");

/*--------Now, we receive data from server--------*/
printf("before receive data\n");
total = 0;
i = t_rcv(ffd,msgr.size,&flag);
if(i > 0) total += i;
if(i < 0) printf("receive err %d\n",t_errno);

gettimeofday(&btimBuff);
while(total != size)
{
    i = t_rcv(ffd,msgr.size,&flag);
    if(i > 0) total += i;
}

gmtime(&etimBuff);

/*----------Compute the delay---------------------*/
start = ((double)btimBuff.tv_sec)*1000 + btimBuff.tv_usec/1000;
stop = ((double)etimBuff.tv_sec)*1000 + etimBuff.tv_usec/1000;
eptime = stop - start;

fp = fopen(REPORT,"a");
fprintf(fp,"\%lu\n",eptime);
fclose(fp);

close(ffd);
sleep(4);
}
Appendix D. UDP with TLI

/*--------------------------------*/
/* UDP TLI: sending program */
/*--------------------------------*/
#include <stdio.h>
#include <tiuser.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stropts.h>

#define DEV_UDP  "/dev/udp"
#define DEV_TCP  "/dev/tcp"

#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 6000
#define MAXSIZE 4096
#define CNTFILE "para"

main()
{
  int   tfd;
  struct t_bind  req;
  extern  int t_errno;
  char   msg[MAXSIZE];
  int    i,j,k,dd,remainder,grainsize,replica,flag;
  FILE   *fp;
  unsigned long size;
  struct t_unitdata *udataptr;
  struct sockaddr_in serv_addr, cli_addr;

  /*-------------------read in control parameters-------------------*/
  fp = fopen(CNTFILE, "r");
  fscanf(fp,"%d %lu %d", &replica, &size, &grainsize);
  fclose(fp);

  /*-------------------Now, carry out the experiments-------------------*/
  for(k = 0; k < replica; k++)
  {
    if(tfd = t_open(DEV_UDP, O_RDWR,(struct t_info *) 0)) < 0)
      perror("server: can't t_open");
/*------Create a TCP transport endpoint--------*/
bzero((char *) &serv_addr,sizeof(serv_addr));
serv_addr.sin_family = AF_INET;
serv_addr.sin_addr.s_addr = htonl(INADDR_ANY);
serv_addr.sin_port = htons(SERV_UDP_PORT + k);

req.addr.maxlen = sizeof(serv_addr);
req.addr.len = sizeof(serv_addr);
req.addr.buf = (char *) &serv_addr;
req.q_len = 5;

if(t_bind(fd,&req,(struct t_bind *)0) < 0)
    perror("server: can't bind local address");

/*------Allocate a t_call stru. for t_unitdata stru.------*/
if((udataptr = (struct t_unitdata *) t_alloc(tfd,T_UNITDATA,T_ADDR)) == NULL)
    perror("server: t_alloc error for T_CALL");

/**********Accept data request**********/
udataptr -> opt.maxlen = 0;
udataptr -> opt.len = 0;
udataptr -> udata.maxlen = grainsize;
udataptr -> udata.len = grainsize;
udataptr -> udata.buf = msg;
if(t_rcvudata(tfd,udataptr,&flag) < 0)
    perror("data request failure");

/**********send the data on UDP endpoint**********/
    dd = size / grainsize; remainder = size % grainsize;
    for(i = 0;i < MAXSIZE;i++) msg[i] = t;

    for(j = 1;j < dd;j++)
    {
    if(t_sndudata(tfd,udataptr) < 0)
        { perror("write error"); exit(0); }
    }

if(remainder == 0) msg[0] = 'e';
    if(t_sndudata(tfd,udataptr) < 0)
        { perror("write error"); exit(0); }

    msg[0] = 'e';
    udataptr -> udata.maxlen = remainder;
    udataptr -> udata.len = remainder;
    if(remainder != 0)
    if(t_snd(tfd,udataptr) < 0)
        { perror("write error"); exit(0); }

    t_close(tfd);
}

/******************UDP TLI: receiving program*********/
/*-------------------------------------------*/
#include <stdio.h>
#include <tiuser.h>
#include <sys/types.h>
#include <sys/socket.h>
#include <netinet/in.h>
#include <stropts.h>
#include <sys/time.h>
#include <time.h>

#define DEV_UDP "/dev/udp"
#define DEV_TCP "/dev/tcp"
#define SERV_UDP_PORT 6000
#define SERV_TCP_PORT 6000

#define SERV_HOST_ADDR "128.173.6.161" /* loki address*/
#define MAXSIZE 4096
#define REPORT "tliudp"
#define CNTFILE "para"

main()
{
    int tfd,i,flag,n,k,j,remainder,replica,grainsize;
    struct sockaddr_in cli_addr,serv_addr;
    struct t_unindata unindata,*rudataptr;
    char msg[MAXSIZE];
    extern int t_errno;
    struct timeval btime_buff,etime_buff;
    double stop,start,avgthr = 0,sdev = 0,thr,dtime;
    unsigned long avgerr,size,total,elptime;
    FILE *fp,*fp1;
    time_t tm;

    /*------------read in control parameters-------------*/
    fp = fopen(CNTFILE,"r");
    fscanf(fp,"%ld %lu %d",&replica,&size,&grainsize);
    fclose(fp);

    /*------------Create the report file--------------*/
    fp = fopen(REPORT,"w");
    fclose(fp);

    /*-------------Now ,carry out the experiments------------*/
    for(k = 0;k < replica;k++)
    {
        if((tfd = t_open(DEV_UDP,O_RDWR,(struct t_info *) 0)) < 0)
            perror("client: can't t_open");

        if_t_bind(tfd,(struct t_bind *) 0,(struct t_bind *)0) < 0)
            perror("client: can't bind local address");

        /*--------Create a TCP transport endpoint --------*/
        bzero((char *) &serv_addr,sizeof(serv_addr));
        serv_addr.sin_family = AF_INET;
        serv_addr.sin_addr.s_addr = inet_addr(SERV_HOST_ADDR);
        serv_addr.sin_port = htons(SERV_UDP_PORT+k);
    }

Appendix D. UDP with TL1 103
/*----------Initialize a unitdata struc. for sending data----------*/
unitdata.addr.maxlen = sizeof(serv_addr);
unitdata.addr.len = sizeof(serv_addr);
unitdata.addr.buf = (char*) &serv_addr;
unitdata.opt.maxlen = 0;
unitdata.opt.len = 0;
unitdata.opt.buf = (char *) 0;
unitdata.udata.len = grainsize;
unitdata.udata.buf = msg;

/*-------------------send a data request-------------------*/
if(t_sndudata(tfd,&unitdata) < 0)
    perror("data request error");

/*--------Allocate memory for t_unitdata struct--------*/
rudataptr = (struct t_unitdata *) t_alloc(tfd,T_UNITDATA,T_ADDR);
if(rudataptr == NULL)
    perror("client: t_alloc error for T_CALL");
rudataptr->opt.maxlen = 0;
rudataptr->udata.buf = msg;
rudataptr->udata.maxlen = grainsize;

/*--------Now, we receive data from server--------*/
total = 0;
i = t_rcvudata(tfd,rudataptr,&flag);
if(i == 0) total += rudataptr->udata.len;
if(i < 0) printf("rece err %d\n",t_errno);

gettimeofday(&btime_buff);
while(msg[0] != 'c') {
    i = t_rcvudata(tfd,rudataptr,&flag);
    if(i == 0) total += rudataptr->udata.len;
    if(i < 0) printf("rece err %d\n",t_errno);
}
gettimeofday(&etime_buff);

/*--------Compute the delay--------*/
start = ((double)btime_buff.tv_sec) *1000 + btime_buff.tv_usec/1000;
stop = ((double)etime_buff.tv_sec) *1000 + etime_buff.tv_usec/1000;
elpume = stop-start;

fp = fopen(REPORT,"a");
fprintf(fp,"%lu %lu\n",elptime,total);
close(fp);

close(tfd);
sleep(2);
}
Vita

Qizhong Chen was born in Shanghai, China on Oct. 28, 1964. He received his high school diploma from the Shanghai Beijiao Middle School in July, 1982. He joined the Computer Science Department at Fudan University, Shanghai, China. He received his Bachelor of Science degree in computer science in July, 1986. He was a graduate student at the same university after his graduation.

He began work towards his Master Degree in Computer Science at Virginia Tech in August, 1989 and completed the requirements for the Master Degree in July, 1991. He is a student member of the ACM.