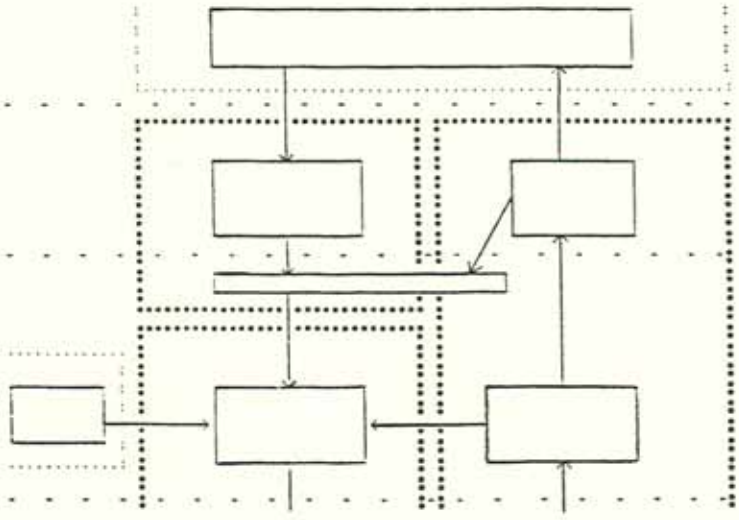
MIT/LCS/TM-233

IMPLEMENTING INTERNET REMOTE LOGIN ON A PERSONAL COMPUTER



Louis J. Konopelski
December 1982

Implementing Internet Remote Login on a Personal Computer

by

Louis J. Konopelski

December, 1982

© Massachusetts Institute of Technology 1982

Funding for this research came from IBM through discretionary funds provided to the M.I.T. Laboratory for Computer Science.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge, Massachusetts

02139

Implementing Internet Remote Login on a Personal Computer

by

Louis J. Konopelski

Submitted to the
Department of Electrical Engineering and Computer Science
on December 17, 1982 in partial fulfillment of the requirements
for the Degree of Bachelor of Science

Abstract

This thesis demonstrates that a desktop personal computer can support an efficient internet remote login implementation with the same protocols used by large mainframes. It describes a project in which the Telnet remote login protocol, along with the supporting Transmission Control Protocol and Internet Protocol were implemented on an IBM Personal Computer. The utility of the implementation depended heavily on the software speed. Strategies discussed to insure quick performance included tailoring protocols to their clients needs, sharing the overhead of asynchronous actions, and sharing data. A natural order in which to process the protocol data was identified, and two control structures were presented that allowed the protocol modules to run in this order. One of the control structures used procedures and processes, while the other used procedures alone.

A full scale protocol was successfully placed in the personal computer. With some foreign hosts, the implementation echoed characters in less than a quarter of a second, and processed a screenful of data in less than three seconds. The protocol software overhead was never the dominating performance bottleneck. The serial line interface limited the character echoing performance while the speed with which the processor could operate its display limited the processing speed of large amounts of data. Memory size was not a significant constraint.

Keywords: internet remote login, personal computer, asynchrony, tasking

Acknowledgments

Many people contributed to the success of the personal computer remote login project. Wayne Gramlich, Chris Terman, John Romkey, and David Bridgham produced a number of software development tools including a compiler for the C language, an assembler, a linker, a C Standard I/O Library, and some Heath 19 terminal emulator routines. David Bridgham, Karl Wright, and John Romkey designed and wrote the low level protocol. David Bridgham additionally spend much time keeping the packet concentrator running through numerous hardware changes, while John Romkey additionally wrote the signal handling routines and the Internet layer code that were modeled after similar routines written by Larry Allen in C for a PDP-11 with a Unix operating system. The TCP, Telnet, and tasking routines drew heavily from implementations written in C by Larry Allen for a PDP-11, which in turn were translated from code written in BCPL for a Xerox Alto by Dr. David Clark.

I am also indebted to Dr. Clark for his ideas on tasking which provided a convenient framework in which to think about protocol development; to Larry Allen, who helped me to iron many philosophical lumps out of this thesis, and to John Romkey, who provided moral support when events did not go as planned.

Most of all, I would like to thank Prof. Jerome Saltzer for his wisdom and insight that kept me out of many tar pits, for his time spent reading the drafts of this document, and for his patience in seeing me through to the completion of my thesis.

Funding for this project came from IBM through discretionary funds provided to the the M.I.T. Laboratory for Computer Science.

Table of Contents

Table of Figures

Table of Tables

Chapter One

Introduction

Personal computers are the most recent outgrowth of the hardware revolution to attract the attention of the public. Their mass market potential allows manufacturers to provide them at relatively low costs, yet they remain significant sources of computational power with a range of applications much wider than video games or simple mathematical calculations. A goal of the Computer Systems and Communications Group [1] at the M.I.T. Laboratory for Computer Science is to show that such machines are powerful enough to exploit the advantages of network communications in ways that were previously reserved for much larger machines. This thesis deals with a subset of this problem, implementing internet remote login on a desktop personal computer.[1] Remote login protocols allow a user on one computer to log onto another computer via a network. They provide a facility for communication that is much more flexible than directly wiring a terminal to a computer, yet is many times faster than a connection across 300 or 1200 baud commercial telephone lines. With appropriate network connections, a computer can be accessed by users nationwide with data transfer rates measured in kilobits or megabits. The primary disadvantage to internetwork communication is equipment cost, but as hardware prices have been falling, this barrier has been crumbling.

Compared with the machines on which remote login protocols usually run, personal computers have smaller memories, narrower data paths, and slower processor speeds. Efficient design is necessary if protocols on personal computers are to have acceptable performance. In the design described by this paper, areas of special efficiency emphasis include copy minimization, data sharing, buffering strategies, asynchrony minimization and

---

[1] Karl Wright, in a companion paper [2], wrote about a different subset of the problem of using personal computers for network communication, namely, implementing a file transfer program on a personal computer.

writing tailored layer implementations to better meet the needs of client layers. An area of particular concern is the overhead of dealing with asynchronous events. This paper proposes a natural order in which to process data generated by asynchronous events, and presents a control structure which efficiently and modularly follows this order.

This thesis shows that a desktop personal computer can support an efficient internet remote login implementation with the same protocols used by large mainframes. It describes an implementation of the Telnet, TCP, and Internet Protocols on an IBM Personal Computer which allows data to be transferred at rates in excess of 5000 bits per second.

The Telnet, TCP and Internet Protocols, the IBM Personal Computer, and the other hardware and software which support the remote login protocol are described in the next chapter. Chapters III and IV describe ways to make the protocol implementations efficient. Tailoring layers to meet the needs of their clients, sharing the overhead of asynchronous action, and sharing data are the subjects of Chapter III. Chapter IV discusses control structure. The notion of a natural order of data processing is developed, and two methods for achieving it are presented. Implementation details are the subject of Chapter V. Testing and protocol evaluation are discussed in Chapter VI, while the final chapter summarizes the results and presents suggestions for improving the implementation and areas for further research.

1.1 Definition of Terms

A number of terms need to be defined before proceeding. If A and B are procedures, they are in the same ***thread of control*** if their local variables are on the program stack at the same time. Another way of phrasing this is that A and B share a thread of control if either A or B calls the other or calls some other program which calls the other. A ***process***, for the purposes of this paper, is a program in execution. (The remote login protocol described herein required three processes: one to process information typed by the user, one to process information from the network, and one to send packets.) Processes do not have separate address spaces. A process is ***blocked*** when it is not executing and when it will never execute

unless some specific (awakening) action is taken. A process is ***awake*** when it is not executing but when it will execute at some point in the future without any additional action being taken. A ***running*** process is a process in execution. ***Scheduling*** means deciding which process to run next.

Interrupt driven code runs in response to an interrupt, and preempts the code that is currently executing. It stores the contents of the registers in memory, and sets up a new stack on top of the old stack. It also ***turns off*** interrupts, i.e., it prevents the microprocessor from preempting it if an interrupt occurs while it is running. When the interrupt driven code is done, it turns interrupts back on, restores the stack and the registers, and allows the program that was running before it to continue execution.

If a second interrupt occurs while code is running in response to a first interrupt, the second will not be processed until the first's code has finished. The event that causes the second interrupt can be lost if the lifetime of the event is shorter than the time taken by the code that processes the first interrupt. In practice, interrupt driven code took less time than the lifetime of most events, so very few events were lost in this way.

Chapter Two

Software and Hardware Choices

The evidence that personal computers can efficiently support internet remote login was built out of particular software and hardware. This section describes the reasons for choosing the particulars.

2.1 The Protocols

The remote login protocol chosen for the demonstration described by this thesis was Telnet, the DOD standard remote login protocol. Supporting it were the DOD standard Transmission Control Protocol and Internet Protocol. These protocols were becoming widely implemented on the various local networks to which the personal computers were attached. They would also allow the personal computers to send packets nationwide via the ARPANET.

2.1.1 The Internet Protocol

The Internet Protocol [3] provides a level of standardization that allows long distance data communications across nonstandardized local networks connected by gateways. It is designed for transferring blocks of data across packet switched networks. To each block of data, the Internet Protocol adds a header that contains information allowing gateways to appropriately send the packet across the local network. Internet does not provide a reliable transmission facility. It drops packets it cannot appropriately process.

The most important function of the Internet layer is addressing. Each source or destination is a host, identified by a fixed length address. If the destination of a packet is a host on the local network to which a gateway is attached, it sends the packet to the host. Otherwise, it forwards the packet to a gateway closer to the ultimate destination. (The mechanism which gateways use to decide where to forward a packet is beyond the scope of this paper.)

The Internet header also carries information to allow gateways to adjust the parameters of the local network to best transmit a packet. Internet allows the sender to rate the importance of a packet on a scale of one to eight. It also allows the sender to specify if the packet requires low delay, high throughput, or high reliability.

A particularly important parameter of a local network is the maximum packet size that it can transmit. Internet provides a facility where packets can be broken into smaller units called fragments and reassembled at their ultimate destination. Each fragment is marked as such and assigned an ID number starting with 0. The last fragment is additionally marked.

Internet provides a ***time-to-live*** field to prevent misrouted packets from being forwarded forever, and a header checksum to insure that the information in the Internet header is correctly transmitted. Optional Internet control fields allow hosts to send timestamps, security information and special routing information. Internet can demultiplex packets among various client layers.

Internet does not provide for data error checking, retransmission of lost data, or flow control. These functions are assumed to be part of higher level protocols. Additionally, Internet makes no guarantee of the order in which it passes packets to its clients, and, except for fragments, assumes that each packet is separate from all others.

2.1.2 The Transmission Control Protocol

The Transmission Control Protocol [4], better known as TCP, makes use of the unreliable Internet Protocol to provide reliable internetwork communications between pairs of processes. It provides an error free, correctly ordered, and bidirectional (full duplex) transmission of streams of data.

To insure reliably ordered data, each process assigns a sequence number to the first byte of data in the stream that it is sending. Processes refer to each succeeding byte of data by the succeeding sequence number (modulo 2^{32}). The receiving process acknowledges data it receives by returning an acknowledgment number which is just the sequence number of the

next byte that it is expecting to receive. When the sending process receives this acknowledgment number, it assumes that the receiving process has correctly received all previous bytes. The sender retransmits any data not positively acknowledged in a reasonable period of time. The sequence numbers, when combined with a TCP length field and a checksum on the TCP header and data, insure reliably ordered, error-free data delivery.

TCP also allows each host to have multiple processes and each process to have multiple connections. When a process wishes to open a connection, it asks the host for a host unique connection ID, called a port number. The process concatenates this ID with the host's Internet address to form a socket number. The socket numbers of two communicating processes uniquely specify a connection. The TCP specification reserves certain port numbers. In particular, it reserves port 23 for the Telnet server process.

TCP provides flow control by having a process specify a ***window*** of the number of bytes of data that it is prepared to receive. This defines the maximum number of bytes of data that the sender may normally have outstanding, i.e., sent but not acknowledged. The window size also implies that the receiver can store and reorder any data in the window arriving out of sequence. ***Out-of-sequence*** data refers to the situation where a receiving process gets the data byte with the sequence number 2001 before it receives the data byte with sequence number 2000. Because TCP promises its client a reliably ordered stream of data, it must buffer data byte 2001 until it receives byte 2000, at which point it may pass both bytes to the client.

TCP generally determines the best time to send data by its own volition. However, it provides a ***push*** function through which the applications layer may tell it to send data immediately. TCP also provides an ***urgent*** function through which the applications layer causes TCP to set an urgent pointer to the data. The urgent pointer specifies the location of the urgent data in the upcoming data stream so that the foreign TCP can process the urgent data quickly.

2.1.3 The Telnet Protocol

The Telnet Protocol [5] provides a bi-directional, eight-bit-byte oriented remote login facility by specifying a ***network virtual terminal*** (NVT) through which terminal-oriented processes communicate. Telnet also provides a mechanism whereby processes can negotiate options different from those provided by NVT. Telnet uses TCP to transmit a stream of bytes which consist of USASCII characters with interspersed Telnet commands.

The network virtual terminal uses USASCII codes. The local terminal is responsible for echoing. NVT is essentially a half duplex device operating in line buffered mode. It normally transmits characters a line at a time unless explicitly told to do otherwise by the user. The Telnet server must send a Telnet ***go ahead*** command when it can not proceed without further instructions from the user. This allows a process with a half duplex terminal to decide when to switch control of the terminal to the user.

Telnet defines standardized commands for certain control functions found on most servers that are often invoked differently. It defines a commands to interrupt running processes, abort output, signal that the server is still running, erase characters, and erase lines. It also provides a mechanism for the server to tell the user to ignore any data buffered between the server and the user.

Terminal options can be changed from those provided by the network virtual terminal by negotiation. Four special Telnet signals, ***WILL***, ***WON'T***, ***DO***, and ***DON'T*** are defined for this purpose. DO indicates a request for the other party to perform a service, DON'T asks the other party to stop. WILL indicates the desire to begin performing a service while WON'T indicates a desire to stop a service. The personal computer Telnet can negotiate a full duplex connection by turning off *go ahead* commands, and it can negotiate remote character echoing.

All Telnet commands are one byte long and are preceded by the ***IAC*** character, (data byte 255). Telnet considers all characters not preceded by IAC to be data. Character 255 is sent by sending IAC twice.

There is a distinction between *user* Telnet and *server* Telnet. User Telnet, when running on computer A, allows a user of A to login to computer B. It reads characters from the keyboard and sends them to the net, and prints characters from the net on the screen. Server Telnet, when running on computer A, allows users on any foreign host to login to computer A. It passes incoming characters to the operating system and writes the operating system's responses to the net. We only wrote user Telnet for the personal computers since we had no current need for server Telnet, and since its implementation would only have delayed completion of the much more useful user Telnet.

User Telnet normally sends all characters that the user types to the foreign host, but it also provides the user with an escape sequence that the user employs to request a Telnet service such as negotiating an NVT option or sending the Telnet command that will interrupt the foreign process.

2.2 The Personal Computers

The IBM Personal Computer was a typical product in the 1982 desktop personal computer market. We chose it for our project because it had a number of useful features:

1. Its memory was not restricted to 64 kilobytes.
2. The documentation was well suited to software development because it included commented ROM listings and many schematic diagrams.
3. High level programming languages and software tools were available for it.

The processor was the sixteen bit Intel 8088 which had an eight bit data path and which ran on a 4.77 Mhz clock. The configuration that we used had 194 KB of primary memory and two disk drives, each capable of holding a 160 KB floppy disk. It also had an asynchronous communications adapter which could connect to an RS-232 line.[2]

---

[2] The *Technical Reference* [6] provides additional technical details on the IBM Personal Computer.

2.3 Software Development Tools

At the time we started this project, we seriously considered two languages for software development: Pascal and C. We would have written and compiled Pascal programs on the IBM Personal Computers, while we had to write and cross-compile C programs on an available PDP 11/45[3] with a Unix[4] operating system, and download the programs to the personal computers. We chose the second option because a Pascal-compatible assembler was not available for the IBM Personal Computer at the time we started this project.

The software tools available to develop programs described in this paper included a compiler for the C language, an assembler, a linker, a C Standard I/O Library, and some Heath 19 terminal emulator routines.

2.4 The Network Interface

We attached the IBM Personal Computers via an RS-232 line to a Digital LSI-11 which acted as a packet concentrator and gateway between the IBM Personal Computers and the V1 ring, a 1 Mbit/sec token ring net with a Unibus[5] interface. From the V1 ring, packets could get to the ARPANET and a number of local networks.

We used a packet concentrator to provide a cost effective way of connecting multiple IBM Personal Computers to the V1 ring, until such time as a local net for the IBM Personal Computers became commercially available. We chose an LSI-11 as a packet concentrator and gateway because we had a software development system for code of this type for it.

The *low level protocol* (LLP), the interrupt driven code that ran on the personal computers and dealt with the serial line is described in detail in Section 5.3.

---

[3] PDP is a trademark of the Digital Equipment Corporation.

[4] Unix is a trademark of the Bell Telephone Laboratory.

[5] Unibus is a trademark of the Digital Equipment Corporation.

Chapter Three

Designing an Efficient Implementation

Perhaps the most noticeable difference between small and large computers from the point of view of the user is the difference in the speed with which similar programs run. The usability of the personal computer protocols depended heavily on the efficiency or quickness of the implementation. Experience with other implementations indicated a number of guidelines to follow to insure the efficiency of the final code. They are presented in this chapter.

3.1 The Meaning of Efficiency for Personal Computer Telnet

Protocols are programs for transferring data. A goal of the personal computer protocol implementation is to transfer data as quickly as possible. For the purposes of this paper, efficiency refers to the time it takes to transfer data. The less time it takes to transfer data, the more efficient the protocol.

Two units of data were identified as being particularly important in the Telnet environment: a single character and a screenful of characters. Packets with a single character of data will be common while Telnet is in remote echo mode. The user will be sensitive to the round trip delay of such a packet. Many user requests will cause the foreign host to send a screenful of data to the user in large packets; the personal computer protocols must also process these screenfuls efficiently. (Some requests will produce more than a screenful of data, but we assume that they will do so a screenful at a time, asking the user for positive response before showing the next screenful. Thus we assume a screenful of data to be the largest amount of uninterrupted data that the foreign host will sent.)

The benefits that the protocols derive from improving the the handling time for these two

types of data are not linear, nor are the strategies for improving them always complementary. The user will find a single character round trip delay of more than a fourth of a second unacceptable. Yet the user will not notice improvements that decrease the delay to less than a fortieth of a second. The protocols should process the 2000 characters necessary to fill the screen as quickly as possible, but they can not set up elaborate schemes for managing a lot of data if it means that they will process slowly packets that contain a single data character.

3.2 Tailored and Specialized Protocol Implementations

One of the justifications for a layered protocol specification is that it allows one protocol to serve several different clients. Unfortunately, a protocol built to serve multiple clients seems to follow Hammer's Law: If it's good for everything, it's good for nothing.

An alternative strategy is to design a protocol implementation to meet the needs of a particular client. This strategy can be applied with two different degrees of severity. A mild use of this idea is to tailor a protocol layer so that while it completely meets its specification, it does not perform equally well with all clients. A more extreme form of this idea is to omit some features of a particular layer that the client of interest does not use. A protocol specialized in this way would not work at all with clients that needed the omitted features. The local experience with tailored and specialized protocols indicates that such protocols run five to ten times faster with particular clients than implementations designed to support many clients.

An example of tailoring is the personal computer TCP strategy for buffering output data which TCP must store in case the data is lost and TCP needs to retransmit it. TCP could have either Telnet or FTP (short for File Transfer Protocol [7]) as its client. FTP almost exclusively fills output packets with hundreds of bytes of data drawn from files. Telnet fills packets with data typed by a human user who types slowly. With Telnet there should never be more than a few bytes of data outstanding, (i.e., typed by the user but not yet acknowledged by the foreign host), while an FTP connection could have thousands of such

bytes. The tailored personal computer TCP uses a single packet to hold outgoing data. This system fully meets the TCP specification, which does not stipulate a minimum number of output packets, but such a system would have abysmal performance with FTP since it would limit the amount of outstanding data to the size of a single packet. It works fine with Telnet, however, where the outstanding data can always fit into a single packet. Tailoring TCP to use a single output packet saves the overhead of dealing with multiple packets. TCP does not have to queue packets or otherwise distinguish packets that might need to be retransmitted, and it does not have to recopy much of the TCP header each time it sends a new packet. In fact, TCP uses the fields of the single outgoing packet as state variables.

In addition to having tailored output data buffering, TCP also has a tailored strategy for buffering incoming out-of-sequence data. TCP bases the out-of-sequence data buffer size on the maximum TCP window size, which it in turn bases on the assumption that the largest amount of data that the user will ever wish to see at once is one CRT screenful. Section 5.6 describes the TCP window and buffer management strategies more fully.

Another example of TCP tailoring is its assumption that the user can never type fast enough to fill the window advertised by the foreign host. TCP treats the case where the user has more data to send than the foreign host has advertised window rather inefficiently so it can process the normal case faster. Again, this would have resulted in poor performance with FTP where the local TCP might often fill the window advertised by the foreign TCP.

The personal computer remote login implementation is specialized in that neither TCP nor Internet is able to demultiplex packets between multiple ports. This specialization does not affect the performance of Telnet, since Telnet requires only one TCP connection. In fact, it enables the protocols to run faster since demultiplexing involves at least some extra overhead. Yet, in this way, the personal computer protocol implementation does not fully meet the specification. TCP could not support FTP which requires multiple simultaneous TCP connections, nor could Internet support multiple protocols simultaneously. (A strategy for fixing these deficiencies is discussed in Section 7.1.

An important observation regarding tailoring and specialization is that neither affects the correctness of the implementation. A foreign computer supporting Internet, TCP, and Telnet should not be able to tell the difference between tailored or specialized implementations and implementations that fully support multiple clients.

3.3 Sharing the Overhead of Asynchronous Action

Any protocol layer which can act as a source of a packet without being prodded by the layer above it exhibits asynchronous action with respect to the higher layer. TCP exhibits asynchronous action with respect to Telnet when it sends packets acknowledging data without being told to do so by Telnet. TCP also retransmits data asynchronously. Telnet shows asynchronous action with respect to the user when it responds to foreign negotiation requests without prodding by the user.

Asynchronous action is important because every packet requires a certain amount of time and computer resources to process irrespective of the data it contains. Processing headers, calling subroutines, and scheduling processes take time. The greater the number of packets in which protocols send a given amount of data, the greater the amount of unproductive processing and the more time the protocols will take to process the data. Protocols should send information in as few packets as possible.

The personal computer protocols combine information resulting from asynchronous action into a single packet whenever possible. The protocols can combine three types of information: the TCP acknowledgment number which is updated in response to the incoming data, any Telnet negotiation characters that Telnet must send in response to data carried in the incoming packet, and all characters typed by the user while the protocols are processing the incoming packet. The protocols combine this information by having the part of TCP that updates the acknowledgment number and the part of Telnet that responds to foreign negotiation requests run before the part of Telnet that processes characters from the user which in turn runs before the part of TCP that sends packets.

3.4 Data Sharing, Buffering, and Minimizing Copies

With the possible exception of process scheduling, protocols tend not to have expensive operations. They mostly spend time in loops doing simple operations over and over. The most common loop in protocols is one which processes the data of the protocols one byte at a time. The number of times a protocol implementation byte processes its data is often a good heuristic measure of its efficiency. Accordingly, a specific goal of the personal computer protocols is to minimize the number of times that data was byte processed.

The typical outgoing data byte is processed five times:

1. A code is placed into a buffer by the IBM Basic I/O System (BIOS) when the user types a character.

2. The Heath 19 terminal emulator gives Telnet the ASCII character corresponding to this special code.

3. Telnet tests the character to see if it is a carriage return or a part of the user's escape sequence. If it is not a part of the escape sequence, Telnet places it in the output packet.

4. TCP checksums the outgoing packet.

5. The low level protocol writes the packet to the net.

The typical incoming data byte is also processed five times:

1. Interrupt driven code places the data coming off the net into an Internet packet buffer in memory.

2. TCP computes a checksum on the incoming data.

3. Telnet reads it and checks to see if it is a carriage return or IAC, the Telnet escape character.

4. The Heath 19 emulator handles it.

5. BIOS places it onto the screen.

TCP sometimes copies data one additional time. If an incoming packet acknowledges only

part of the data in the currently held outgoing packet. TCP copies each byte of unacknowledged data to the start of the data area. Because the amount of data that TCP so copies is usually none or small, the time it spends on these copies is much less than the time that it would need to deal with multiple output packets.

Notice that the protocols are never automatically copy packets across protocol layer boundaries. For example, no layer ever copies data specifically for the Internet protocol layer--either TCP or the low level protocol always pass Internet a pointer to a packet.

Chapter Four

Tasking: A Modular Way of Coping with Asynchrony

Protocol implementation is not a well understood science. One cannot turn to the relevant chapter of a textbook to see how to best do it. As the number of TCP and Telnet implementations in the world grows, problems with the current way of doing things become more obvious, and possible solutions suggest themselves more strongly. This chapter describes tasking, a way of organizing protocol control structure using a combination of procedures and processes that seems to meet the requirements of protocols more naturally than traditional methods of organization.[6]

4.1 The Need for Multiple Threads of Control

This section presents the reasons that multiple threads of control seem desirable in layered protocol implementations.

4.1.1 Modularity Through Layering

Writing a program that can reliably send and receive characters and special commands for the user by twiddling bits on an unreliable network is a large task. Protocol designers have modularized this task by specifying layers of protocols, each of which performs a certain subfunction of this task.

A significant advantage of layered modularization is that it limits interaction between

---

[6] The particular form of tasking used here is the idea of Dr. David Clark and will be fully described by him in a future publication. It is described here because it provided a convenient way of organizing the IBM personal computer protocols. The tasking package presented here does not necessarily accurately or completely implement tasking as envisioned by Dr. Clark. This paper is not meant to be the final word on tasking or a summary of the reasons for using it.

protocol layers. Each layer must deal with only two other layers: the layer immediately above it and the layer immediately below it. The Telnet layer does not need to worry about the low level protocol, while the TCP layer does not have to deal with the user.

Violating modularity sometimes seems necessary or desirable in order to improve performance. While each such change may produce a local efficiency, the sum of many such changes will lead to code that is difficult to understand and to modify, and which may have performance problems in the large that no one really understands. The next two sections describe some characteristics of protocols that make them especially vulnerable to creeping unmodularity.

4.1.2 The Natural Order of Data Processing

Each protocol layer has a function. Examining these functions reveals a natural order in which to process data between the network and the user. Incoming data should first be processed by Internet, then by TCP, and finally by Telnet. There is not a one-to-one correspondence between the number of packets that Internet receives, the number of packets TCP receives, and the number of times Telnet and the user receive data. If Internet finds that the Internet header has a bad checksum, it will not pass the packet to TCP. If Internet passes a packet to TCP that contains only an acknowledgment for data and no new data, TCP will not pass anything to Telnet. Similarly, if Telnet receives only a negotiation request, it will not pass anything to the user.

The opposite situation occurs with outgoing data from the user. It should first be processed by Telnet, then by TCP, then by Internet, and finally passed to the network. Again, the fact that one protocol layer runs does not imply that the next layer in the natural order will also run. The user may ask Telnet to interpret all future characters in a different way, and TCP will not immediately run. TCP may buffer data from Telnet unless explicitly told not to do so, and Internet may not run.

4.1.3 Asynchronous Events, Layers, and Threads of Control

While the order in which layers should run to process data is straightforward, coming up with a control structure that both follows this order and preserves modularity is difficult. Procedure calls seem too inflexible to do this.

To illustrate, suppose Telnet is the top level procedure and that the user passes data to it that must immediately traverse the network. Telnet calls TCP, and passes TCP the data with instructions to send it immediately. TCP formats a packet containing the data and calls Internet which in turn fires up the low level protocol and sends out a packet. After sending the packet, the low level protocol returns to Internet which in turn returns to TCP which returns to Telnet. Telnet then waits until the user has more information to send.

What now happens if a packet acknowledging the data comes in over the net? Internet and the low level protocol need to run, but control remains in Telnet. We could give Telnet knowledge of when a packet comes in from the net so it can call TCP which in turn can call Internet, but then Telnet would have knowledge of the network which violates layered modularity. Also, for incoming packets Telnet would be calling TCP which would call Internet--exactly the opposite of the natural order described above. To achieve the natural order, procedures that handle asynchronous data must not share threads of control. Because asynchronous data arises in multiple layers, and because layers cannot share procedures, there must be at least as many threads of control as there are layers that handle asynchronous events.

All Telnet, TCP, and Internet implementations must deal with a minimum of three asynchronous events: characters from the user, packets from the network, and retransmission timer time-outs. Because a different protocol layer handles each of these, all modular implementations of Telnet, TCP, and Internet must have at least three threads of control: a thread with a Telnet procedure as the top level procedure that processes characters from the user, a thread with a TCP procedure as the top level procedure that runs when the retransmission timer goes off, and a thread with an Internet procedure as the top level procedure that processes incoming packets.

The protocol implementor may desire to create other asynchronous events as well. Section 3.3 showed that the part of TCP that acknowledges data by sending a packet with an updated acknowledgment number should wait until the part of Telnet that handles data from the user has run. Delaying data acknowledgment effectively causes it to become an asynchronous event because no procedure can naturally call the TCP procedure that acknowledges data. The part of TCP that processes the incoming packet can not call it--this part of TCP gives up control to give Telnet an opportunity to run. If Telnet has data to send from the user, it will call TCP again, but Telnet should not have to call a TCP procedure if the user does not type any characters. Thus procedures in the TCP layer must handle two asynchronous events: the retransmission timer and data acknowledgment. If each event were processed by a separate procedure, two threads of control would be needed by the TCP layer, one to retransmit packets and one to send packets acknowledging incoming data. To avoid multiple threads of control in the TCP layer, we used a single procedure sent packets in response to both events. In fact we made a further simplification and had a single procedure send all outgoing TCP packets. Because there is only one output packet, (see Section 3.2), writing a procedure that handles all output packets was easy.

Placing the TCP procedure that sends packets in its own thread of control also provides a convenient way of sharing the asynchronous action overhead that results when Telnet responds to a foreign negotiation request. Telnet calls TCP with some data to send, but TCP does not send it immediately. Instead TCP places it in the same packet as the updated acknowledgment number (that acknowledges the characters which formed the foreign negotiation request), and allows the part of Telnet which handles characters from the user to run.

4.1.4 The Special Needs of Real Time Events

A characteristic of asynchronous events is that they may occur at almost the same time. This can be a problem if the events have a lifetime. If the routine responsible for processing an event can not do so within the event's lifetime, then the routine will lose the event. Examples of events with lifetimes are the signal produced when the user presses a key on

the keyboard and a byte coming in over the serial line. Unfortunately the amount of time needed by the personal computer protocols to completely process such events is much longer than their lifetimes. If two events occur at almost the same time, the second will cease to exist before the protocols completely process the first. To avoid this problem, the personal computer protocols use interrupt driven code to buffer realtime events, and code not driven by interrupts can then process the buffered events at its convenience. Interrupt driven code runs when the user presses a keyboard key, when a byte comes in off the serial line, and in response to a timer interrupt.

4.2 Implementing Multiple Threads of Control

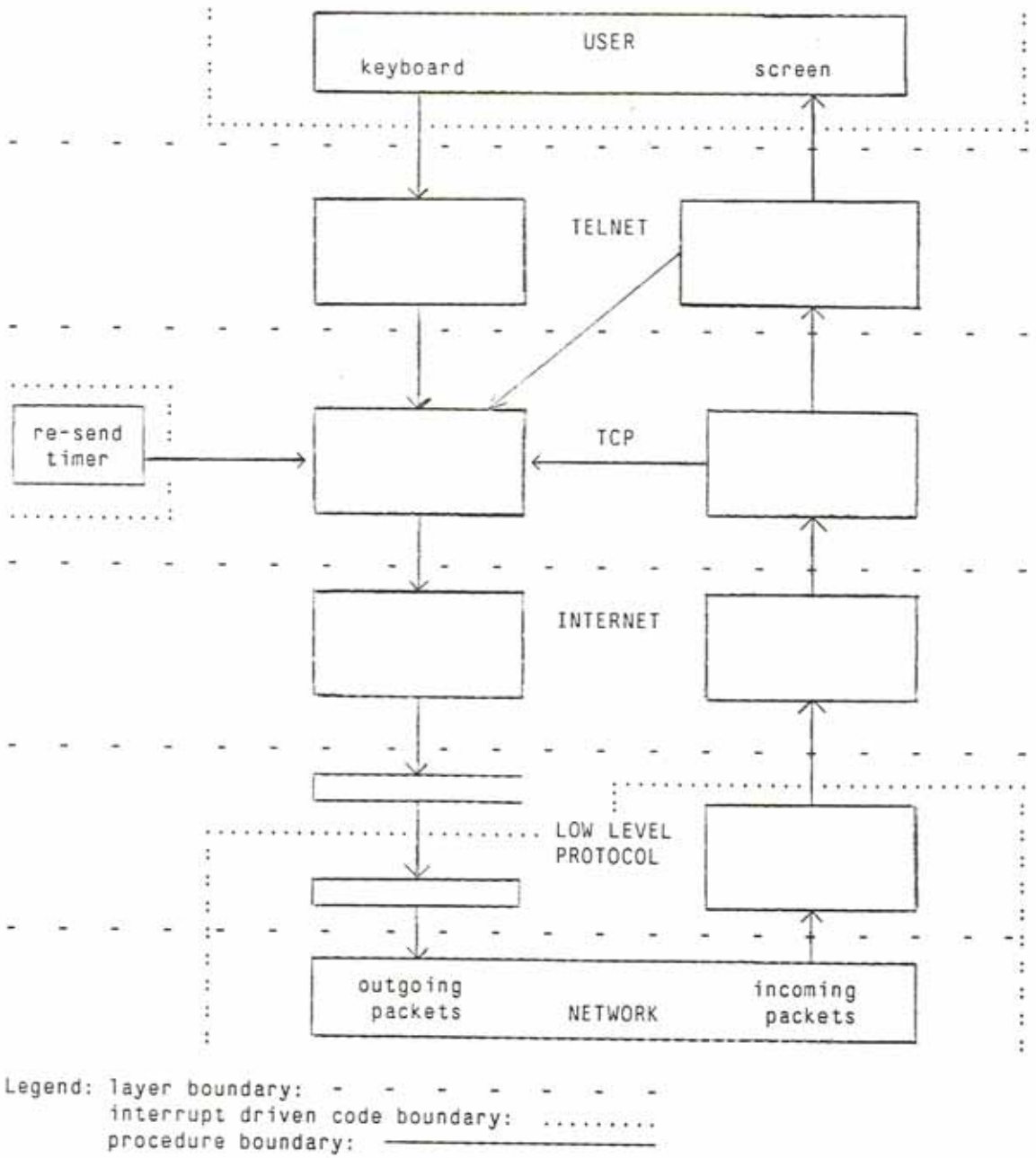
Figure 4-1 summarizes the natural order of data processing described in the previous section. Broken lines indicate layer boundaries. Dots surround interrupt driven code. Blocks indicate procedures inside of layers. Data is first processed by a procedure from which an arrow points and then by the procedure to which the arrow points. This section describes three ways of structuring the multiple threads of control that will process data in the natural order.

4.2.1 Layers as Processes

The traditional way of constructing multiple threads of control is to make each layer a separate process. Because processes can run in any arbitrary order, they can also run in the natural order. Each layer has the potential to be the first to process a piece of asynchronous data. For outgoing packets, the user's keystroke awakens the Telnet process which in turn awakens the TCP process which in turn awakens the Internet process. An incoming packet awakens Internet which awakens TCP which in turn awakens Telnet. A retransmission time-out awakens TCP without awakening Telnet.

In practice, however, scheduling processes in the natural order can be extremely difficult. A simple scheduling system where processes run in the order in which they are awakened does not work well. For example, if an incoming packet arrives over the net and awakens

Figure 4-1:The Natural Order of Data Processing



Internet between the time Telnet gets a character from the user and the time that it awakens TCP to send the character, then Internet will run after Telnet but before TCP. (The low level protocol processes packets coming off the net at interrupt level.) Running Internet between Telnet and TCP can result in unnecessary delay in sending the data that the user types.

An unpredictable order of data processing can cause more serious problems than delaying information. It can make sharing certain data between layers impossible. If the protocols follow the natural order of data processing, Telnet, TCP, and Internet can all use the same outgoing packet. But with an unpredictable order, Telnet could run between TCP and Internet. Telnet could then modify the packet after TCP put a checksum on it, causing the foreign host to reject the packet. Multi-process systems that allow processes to run in an arbitrary order are also difficult to debug.

A more complicated scheduler involving some sort of priorities would produce the natural order, but using it would increase the overhead of scheduling. Increasing the overhead of scheduling is detrimental to performance when sending or receiving Telnet data requires three layers to run, and thus that three processes be awakened, scheduled, and run. In addition, priorities are not modular because to choose the appropriate priority for a particular purpose, a process must have implicit knowledge of how all the other processes will use priorities.

4.2.2 A Procedure-based Scheduler

An alternative scheme would be to make a scheduler the top level procedure and implement layers as a collection of procedures. The scheduler would monitor the asynchronous events, and it would call a procedure in the appropriate layer when an event occurred. The scheduler would effectively act as a switching station between asynchronous events and the procedures that need to process such events, effectively establishing different threads of control for each such procedure. When the user typed information, the scheduler would call Telnet. If necessary, Telnet could then call TCP which in turn could call Internet.

When information arrived over the network, the scheduler would call Internet, and if necessary, Internet could call TCP which in turn could call Telnet. When the retransmission timer went off, the scheduler would call TCP which in turn would call Internet. Thus control would follow the natural order of data processing.

The advantage of this implementation lies in its efficiency. The procedure-based scheduler passes control from itself to a layer and from one layer to another via procedure calls rather than by scheduling and running a new process. Running the layers to process an incoming or outgoing packet requires three procedure calls. For example, when a packet comes in off the net, the scheduler calls Internet, Internet calls TCP, and TCP calls Telnet.

The scheduler, as we described it, is very unmodular. It must know about the net, it must know about the user, it must know about the retransmission timer, and it must know about procedures in all the layers. We could, however, construct a simpler scheduler which is not as unmodular. It would manage a circular list. Each element of the list would have an event flag and a pointer to a procedure. The scheduler would look at each element of the list in round robin fashion. (Round robin scheduling allows the protocol layers to share packets produced by asynchronous action. See Section 3.3.) When the scheduler finds an element with its event flag set, it reset the flag and calls the associated procedure. Each layer that needs to run a procedure in response to an event would give the scheduler a procedure pointer, and would get an event flag pointer after the scheduler added a new element to the list. The layer could then give the flag pointer to the code that handled the event. When the event occurred, the code that handled the event would set the flag. The scheduler would thus have no particular knowledge of either events or the nature of the procedures that respond to events.

We seriously considered the procedure-based scheduler as an implementation strategy for the personal computer protocols; however, we chose to use the tasking package described in the next section. Section 4.2.4 describes the reasons we chose tasking over the procedure-based scheduler.

4.2.3 Tasking

Instead of having a procedure-based scheduler call the top level procedure of the appropriate thread of control in response to an asynchronous event, we could place each such procedure in a separate process. Each process would have its own stack, and the top level procedure in each thread would be at the top (base) of a stack. The interrupt driven code that runs in response to an asynchronous event would awaken the process as well as buffer the event. When the process ran, the top level procedure would handle the event, and it could then call procedures in other layers if necessary.

The remote login implementation needs three processes to make such a system work--one for each thread of control. The variables of the top level procedure of the thread of control sit on top of the process' stack. One process handles characters from the user, and has a Telnet procedure at the top of its stack; one process handles outgoing packets, and has a TCP procedure at the top of its stack; and the final process handles incoming packets from the network, and has an Internet procedure at the top of its stack. As a matter of convenience we will respectively refer to the three processes as the user process, the send process, and the network process.

In this mixed procedure and process system, a protocol layer consists of a collection of procedures which can run in various processes. To hide the constitution of one layer from another, layers may use only procedure calls for interlayer communication. Interprocess communication occurs inside of layers. For example, suppose that the user typing a character causes a Telnet procedure to run in the user process, and that Telnet would like to send this character across the network. Telnet cannot awaken the send process directly, since the top level procedure in this process is a TCP procedure. Instead, Telnet must call a TCP procedure in the user process. This TCP procedure will place the character in the output packet, and awaken the send task. Because a TCP procedure awakens the process that has a TCP procedure as its top level procedure, the interprocess communication occurs within a single layer.

A procedure may run in more than one process. For example, the TCP procedure that

awakens the send process and runs in the user process when Telnet wishes to send data from the user also runs in the network process when Telnet calls it to send characters in response to a foreign network virtual terminal negotiation request.

We will call this particular way of combining procedures and processes *tasking*. A *task* is a combination of a process and the procedures that run in the process. A tasking module contains all the routines that manage tasking, in particular, the routines which allow tasks to go blocked and to awaken other tasks. The tasking module manages a circular list of task control blocks. Each task control block has a flag and a pointer to a task stack. One task could awaken another task by setting the flag associated with its task control block. When a task blocks by calling a tasking module blocking routine, the tasking module will look at each task control block in a round robin fashion. If the flag of a task control block is set, the tasking module will next run the process that uses the associated stack. A task resumes control from the point where it became blocked.

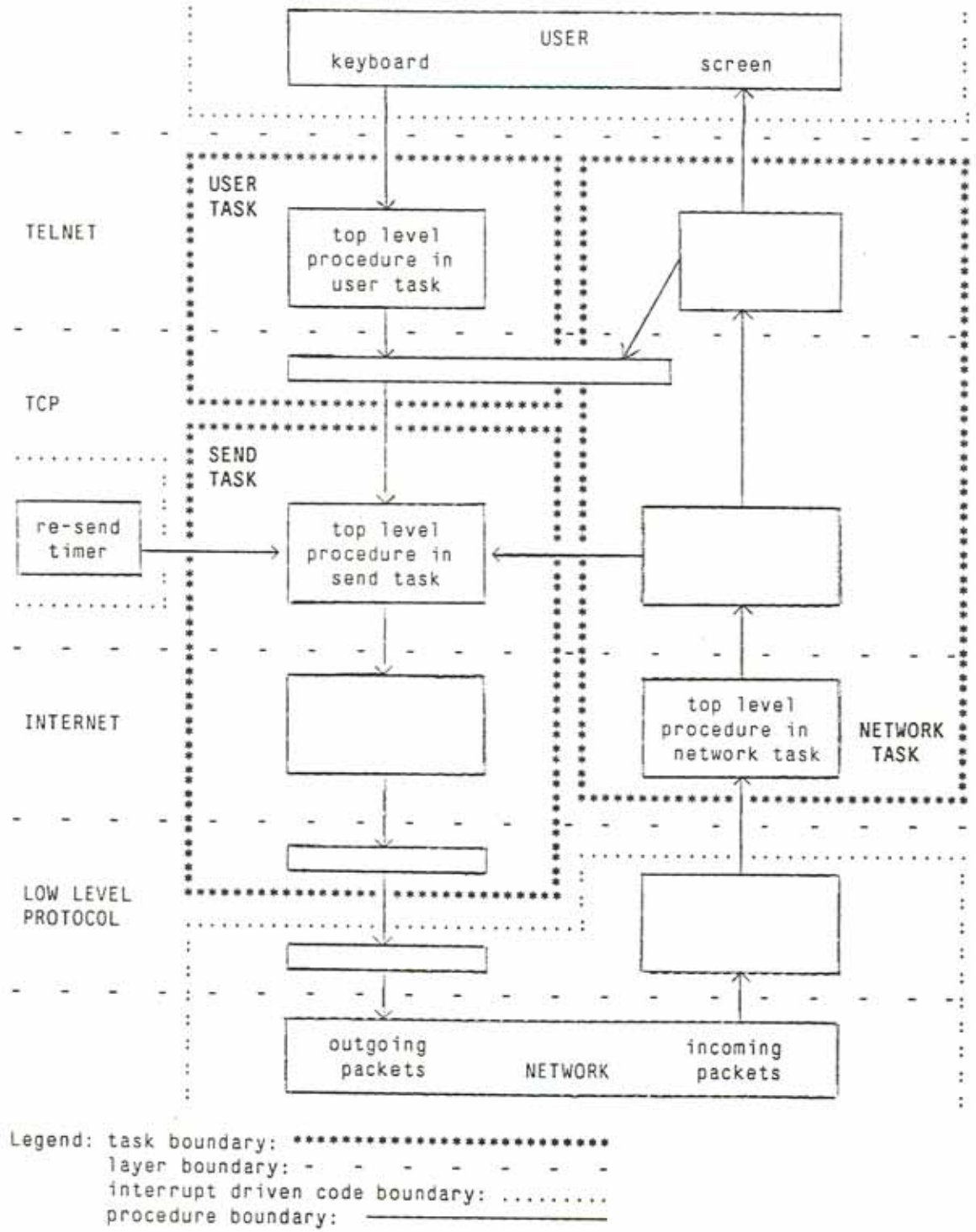
Figure 4-2 shows the task boundaries in the personal computer remote login protocol. The interrupt driven code that runs in response to an incoming packet calls an Internet routine which awakens the network task. When this task runs, Internet is the top level procedure, and can call TCP. TCP in turn can call Telnet, and it can produce an acknowledgment and awaken the send task. When the user task runs in response to a keystroke, Telnet is the top level procedure. If Telnet wishes to send data it calls a TCP routine that awakens the send task. The TCP procedure that is the top level procedure in the send task can then call Internet. The Telnet routine that responds to foreign network virtual terminal negotiation requests and the interrupt driven code that runs in response to a clock interrupt can also call TCP routines in order to awaken the send task.

4.2.4 Procedure-based Scheduling Versus Tasking

Either the procedure-based scheduler or tasking could have formed the basis of an efficient remote login protocol. This section presents the reasons we chose tasking.

A simple procedure-based scheduler would have been somewhat more efficient than the

Figure 4-2:Task Boundaries



tasking package that we actually used. We wrote a simple procedure-based scheduler like the one described in Section 4.2.2, and used it to run some vacuous procedures, (procedures that merely set the event flag of the next procedure to run before they returned.) Swapping control from one vacuous procedure to another in a different thread of control took about 70 microseconds. (Swapping control involved setting an event flag, returning to the scheduler, deciding which procedure to run next, and calling it.) With tasking, swapping control between vacuous procedures in different processes takes about 135 microseconds. (Swapping control with tasking involves setting an event flag, calling a block routine in the tasking module, deciding which task should run next and running it.) Thus the decision to use tasking added 65 microseconds to the cost of processing an incoming packet and 200 microseconds to the cost of processing an outgoing packet. (With tasking both the user task and the send task need to run in order to send an output packet with data from the user.)

An additional advantage to the procedure-based scheduler is that its implementation does not require any assembly language programming.

The primary advantage to tasking is that it is more flexible than the procedure-based scheduler in that it conveniently allows procedures to block at any point and to later continue execution in exactly the same place and with exactly the same state. This is not the case with the procedure-based scheduler which requires that the top level procedure in one thread of control return before the scheduler can call a procedure in a different thread of control.

For example, assume that in the send task, Internet finds that it is missing a resource that it needs to send a packet. It can arrange for the resource's manager to awaken an Internet procedure when the resource becomes available, and then it can block. Internet in a procedure-based scheduler system, however, would have to return to TCP which would have to return to the scheduler before a procedure in a different thread of control could run. Further, re-invoking Internet when the resource becomes available is not as straightforward as with tasking. TCP should not be called in response to an event which concerns Internet, yet TCP may wish to run after Internet has successfully sent the packet. Resource processing may thus need its own thread of control.

The three threads of control in the personal computer protocols generally run to completion in response to asynchronous events, and do not have to block waiting for resources. Thus the ability of the tasking package to stop threads of control in multiple places is not an advantage in the personal computer remote login environment. Telnet, TCP, and Internet would run slightly quicker if implemented with a procedure-based scheduler than if implemented with tasking, but this difference would probably not be noticeable to the user. We chose to use tasking rather than the procedure-based scheduler for two reasons: 1) many other protocol implementations that our group supports use tasking, and putting tasking on the personal computer protocols preserves some consistency among the implementations, and 2) the added flexibility of tasking might come in handy when implementing future protocols on the personal computers such as the Simple Mail Transfer Protocol [8].

Chapter Five

Implementation Details

This chapter presents the details of the IBM personal computer remote login protocols.

5.1 Signals

Signals provide a modular interface between interrupt driven code and the tasking package. The signal routines written for the IBM Personal Computer are loosely modeled on the Unix signal package. A program can call the signal manager with an event code and a procedure pointer as arguments. The signal manager arranges things so that when an asynchronous event occurs, the interrupt driven code that handles the event calls the procedure. The personal computer protocols use two signals: a signal that the low level protocol has received a packet from the net, and the retransmission timer signal. They did not use a signal for the keyboard because the interrupt driven code that buffers characters from the keyboard is a part of the IBM Basic I/O System, and writing such code would have involved changing BIOS. Instead, a procedure in the user task polls a buffer for characters from the user.

Since interrupt driven routines call procedures passed to the signal manager, these procedures run while interrupts are turned off, and they thus must be short. For example, TCP asks the signal manager to arrange to have the interrupt driven retransmission timer routine call a TCP procedure when a particular amount of time has passed. This TCP procedure merely awakens the send task--a very quick operation. Notice that this meets the restriction that interprocess communication should occur within a single layer. The send task has TCP code at its top level, and the procedure that awakens this task is a TCP procedure.

5.2 Tasking

Tasking is a way of organizing procedures and processes to form protocol layers that both preserves layer modularity and efficiently processes the asynchronous data of the layers. A task is a process and a collection of procedures that run in the process. The procedures are often from different protocol layers. Tasks cannot preempt one another although tasks can be preempted by routines which are interrupt driven. A task stops running when it calls a blocking routine in the tasking module. Tasks execute in series. Tasking is not a way of simulating parallel processing.

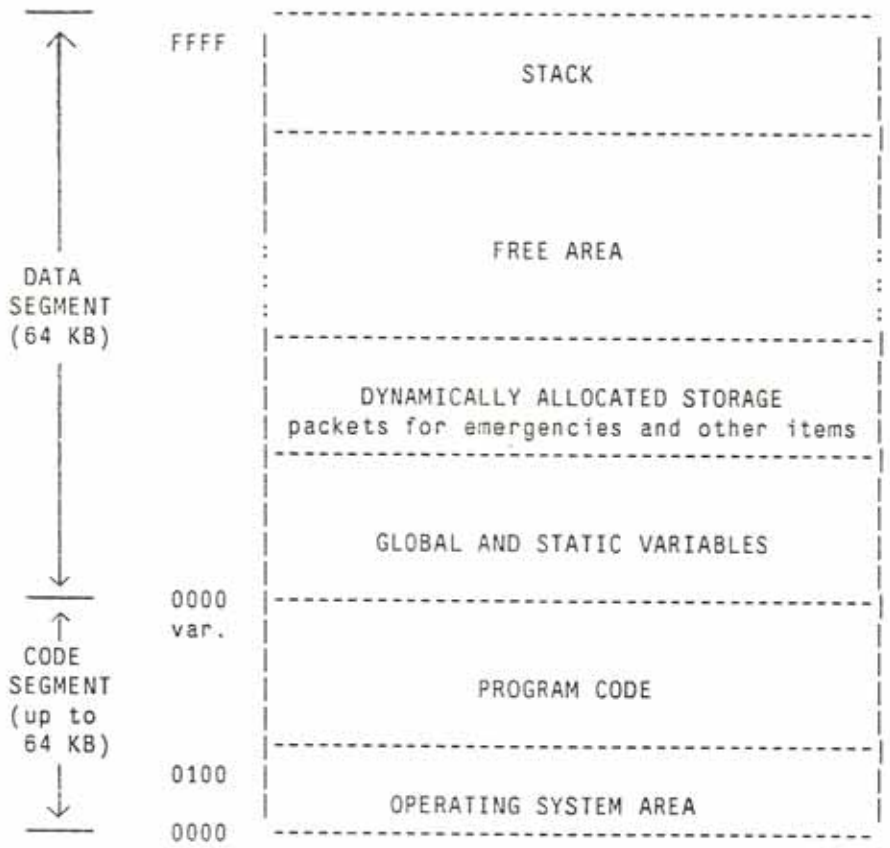
Disciplined task programming requires that one task may awaken another only if the two tasks share state, i.e., internal to a protocol layer. For example, one task must at least know about the task control block of another task in order to awaken the other. To hide the constitution of one layer from another, only procedure calls may be used to pass control from one layer to another.

Figure 5-1 shows the memory organization of the IBM Personal Computer when a typical program runs. The IBM Disk Operating System reserves the first 100 bytes (hex) of the code segment for itself. The program code resides immediately above this. The data segment starts after the program code. At the bottom of the data segment is space for the global and static variables. At the top of the data segment is the program stack which grows downward. Between the stack and the global variables is a free area in which programs can dynamically allocate storage.

Figure 5-2 shows memory organization when tasking runs. It is exactly the same as Figure 5-1 except that there are additional objects in the dynamically allocated storage area, namely task stacks and task control blocks (TCBs). Each task stack is organized like the original program stack. A task control block contains a task's stack pointer, an event flag, and a next TCB pointer.

Task control blocks form a circular list chained by next TCB pointers. An initialization routine in the tasking module creates a task control block with a stack pointer pointing to

Figure 5-1:Memory Organization on the IBM Personal Computer



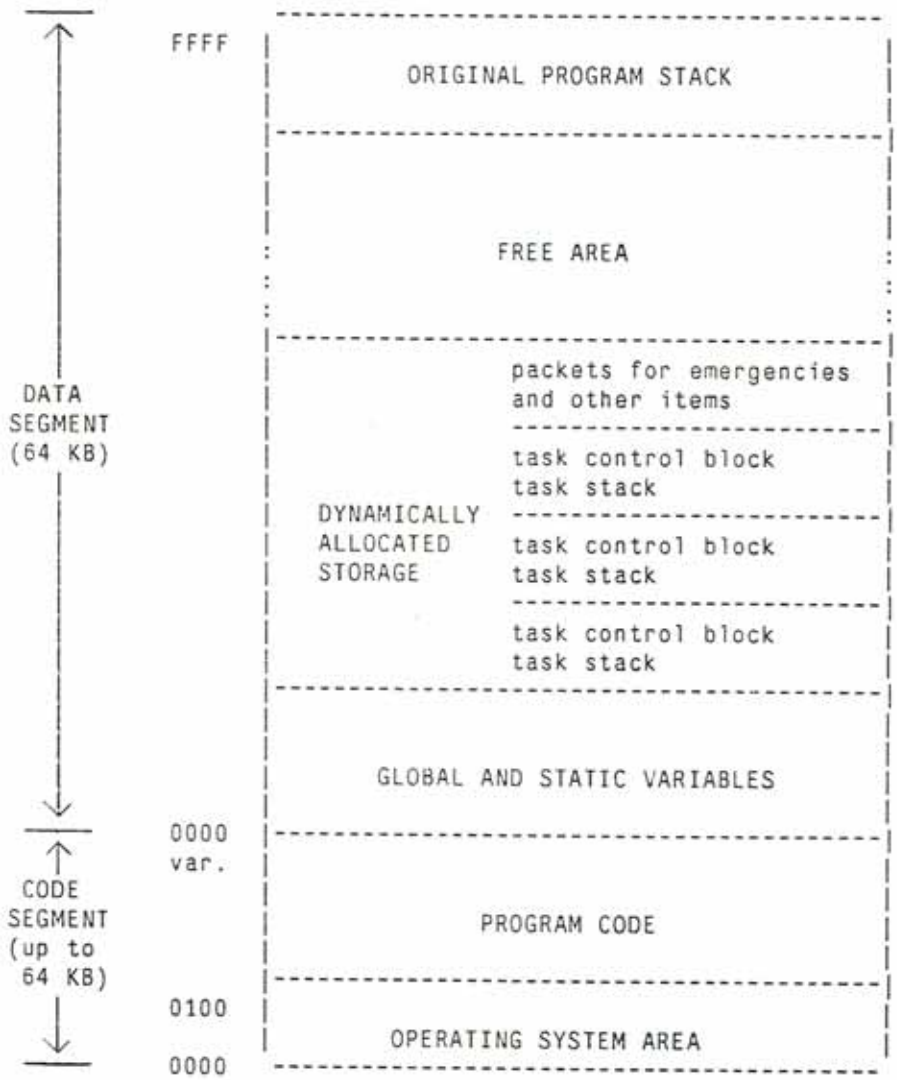
NOTES: Items are not to scale.

var. = variable

the original program stack and a next TCB pointer pointing back to the newly created TCB itself. The initialization routine returns a pointer to this task control block.

Another routine in the tasking module creates additional task control blocks and task stacks. This routine takes a pointer to a task control block and a pointer to a procedure as arguments, and returns a pointer to a new task control block. It places the new task control block in the circular task control block list after the TCB specified by its argument, and initializes the new task control block so that it points to the newly created stack. It also initializes the stack so that when the task runs for the first time, the task starts executing the procedure passed in as an argument. The maximum stack size for a task stack is fixed at the time this routine creates the stack.

Figure 5-2:Memory Organization with Tasking



NOTES: Items are not to scale.

var. = variable

Although the tasking module allowed us to create and free task structures dynamically, in practice we only allocated tasks in the initialization routines and never recycled them. Thus in use, task control blocks and stacks were more static than dynamic.

A task blocks by calling a routine in the tasking module. This routine examines succeeding elements of the list of TCBs, starting with the TCB pointed to by the next TCB pointer of the current task, until it finds a TCB with its event flag set. The task with this TCB is the next task to run. The routine resets the task's event flag, and pushes onto the stack the local variables pointer and a pointer to the line of code immediately after the blocking routine call. This line of code would normally be the next line to execute when the blocking routine returns, but the blocking routine loads the stack pointer of the new task into the stack pointer register of the 8088. On the return, the 8088 loads the program counter with the code pointer on the stack, but since the stack pointer has been changed to point to the stack of the new task, the program counter points to a line of code in the new task. Thus execution in the new task starts with the piece of code following the blocking routine call that caused the new task to block.

The procedure in the tasking module that awakens tasks, when called with a pointer to a task control block as an argument, sets the event flag of the task control block. The routine that blocks tasks resets this flag before it manipulates stacks. A task can awaken itself by calling the routine that awakens tasks with a pointer to its own task control block, thus assuring that it will run again.

The tasking routines are short as well as efficient. They take up about three pages of C code and an additional page of 8088 assembly language routines.

5.3 The Local Network Protocol

The local network protocol between the personal computers and the packet concentrator is called LLP, short for low level protocol. It defines four special characters: *REQ*--request to send a packet, *ACK*--request acknowledged, *END*--end of packet, and *ESC*--an escape character used to send the ASCII characters displaced by the special characters.

LLP is bidirectional. If one machine wishes to send a packet to the other, it sends a REQ. The other machine responds with an ACK when it is ready to receive the packet. The first

machine then sends the packet followed by an END. If the other machine is sending a packet when the first machine sends a REQ, the ACK can be embedded in the packet; the first machine must remove it.

If the wait between REQ and ACK or the wait between any two data characters is too long, LLP assumes that the other machine has crashed. The receipt of a REQ in a data packet indicates that the end of the previous packet has been lost, and LLP discards the packet. LLP also discards unexpected characters.

Most LLP routines are interrupt driven. On an incoming character, these routines check their state to find the character's context. REQ causes LLP to get a packet from the packet manager routine (described in the next section). If LLP can not get a packet or is otherwise unable to process a packet, it will not send an ACK. An END causes LLP to hand a packet to the packet manager as good. A REQ causes LLP to give the packet to the packet manager to be recycled.

One routine in the Low Level Protocol that sends packets is not interrupt driven. Internet calls this routine when it wishes to send a packet over the net. This routine adds a four byte local header to the packet, causes the interrupt driven routine that sent characters to send a REQ, and waits for an ACK. If the ACK does not come within two seconds, the routine reports failure to the Internet layer. If the ACK comes, it causes the interrupt driven routines to send the rest of the packet, and returns to Internet reporting success. This routine waits for the interrupt driven routines to finish sending the outgoing packet before it returns to Internet by running a loop that waits on a flag which the interrupt driven routines set when they have finished sending the packet. The interrupt driven routines that send packets preempt this wait loop. LLP does not return control to Internet sooner, because to do so might allow a higher level protocol to modify the output packet, and cause the packet to have an incorrect checksum.

5.4 The Network Initialization and Packet Manager Routines

The network initialization routine creates two packet queues called the used queue and the free queue, and thirty-two maximum-sized empty packet buffers which it initially places in the free queue. When a routine needs an empty packet buffer, the packet manager provides one from the free queue. The used queue contains incoming packets which LLP has received but the higher level protocols have not yet processed. LLP calls the packet manager to place good packets on the used queue. The Internet procedure that runs in the network task, calls the packet manager to pick up packets from the used queue. When the network task finishes processing a packet it gives the packet back to the packet manager which places it back on the free queue. Packets need not be on either queue. For example, the incoming packet currently being processed by the network task and the output packet are on neither queue.

The network initialization routine also sends a special address request packet to the packet concentrator. The packet concentrator responds with the Internet address of the personal computer making the request. The network initialization routine places this address in a global variable in the I/O library where that all layers can access it. For example, both TCP and Internet need to know the personal computer's address to correctly process the checksums of incoming packets.

5.5 Internet

The actual Internet implementation differs significantly from the model presented in earlier chapters. The implementation ignores Internet's optional fields, it does not do fragmentation or reassembly, nor does it act as the top level of the network task. Instead, TCP acts as the top level, and calls Internet on an incoming packet signal.

The Internet implementation was originally written to support the User Datagram Protocol [9] and the Trivial File Transfer Protocol (TFTP) [10]. TFTP flow control is simple enough not to require tasking. Reassembly, etc., were left out to get TFTP running as quickly as possible to facilitate downloading programs to the personal computers from the PDP 11 on

which they were compiled. To facilitate debugging, we used the same copies of Internet and LLP for both TFTP and Telnet.

An Internet open connection request takes as arguments a protocol number (in this case TCP's number), and a foreign host address. It returns a connection ID. To send packets, TCP calls Internet with the connection ID, a pointer to the packet it wants sent, and the length of the packet. Internet sends the packet to the correct foreign address with the appropriate protocol number. On an incoming packet signal, TCP calls another Internet routine with a connection ID as an argument, and this routine returns a packet if the incoming packet had a good Internet header checksum, if it was from the correct foreign host, and if it was for the TCP protocol. For the purposes of modularity, Internet also provides routines that allocate and free packets, which pass the requests through to the packet manager.

While the Internet implementation is limited, it is complete enough to support connections to all hosts on the M.I.T. networks which support Telnet. It also provides a complete layer interface which effectively hides low level details from TCP. Nevertheless the Internet implementation is incomplete and upgrading it is high on the list of improvements suggested in the final chapter of this thesis.

5.6 TCP

TCP provides an initialization routine that, when called by the Telnet routine that initialized tasking, sets up the send task and the network task. (If Internet had been the top level program in the network task, TCP would have called an Internet initialization routine which would have set up the network task.) The order of tasks in the list of task control blocks is thus the user task, followed by the send task, followed by the network task.

TCP also provides Telnet with routines which will open or close TCP connections, a routine which puts data in the output packet, a push function, a routine to send urgent data, and a routine that prints connection status statistics. These routines run in the user task. The open

routine, close routine, push function , and urgent data routine awaken the send task. The urgent data routine sets the urgent pointer to the data.

A TCP routine is the top level procedure in the send task. It always sends a packet when it runs, and it assumes that the TCP routine that awakened it has appropriately modified the output packet. It calls an Internet routine to send packets, and it calls a Telnet routine if the foreign host does not respond to a request to open a connection within a certain period of time or when the foreign host wants to close the connection. It also passes a pointer to its task control block to the signal handler so that it will be awakened when the retransmission timer goes off. TCP does not have to worry about which packet to send or retransmit as there is only one. All outgoing information is kept in a single output packet. TCP always sends the entire output packet unless the foreign window is smaller than the amount of data in the packet, a situation that never occurs in a normal Telnet connection because the user types so slowly. If it did occur, TCP would call the Internet routine that sends packets with a shortened packet length field as an argument, and the Internet routine would only send as much of the packet as the length field implied currently contained information. (Fortunately, data comes at the end of a packet. This scheme would not have worked if header came at the end of a packet, and Internet truncated part of the header rather than data.) TCP would try to send the data again when the retransmission timer timed out, which would hopefully be after the foreign host had enlarged the foreign window.

The top level TCP routine in the send task is also responsible for managing the window. It upgrades the window only after the window is at least half used up, and it always updates it to the maximum window size. This prevents ***silly window syndrome***, the phenomenon whereby the foreign host sends its data in small packets because the local host enlarges the window by only a small amount [11]. The maximum size of the window is a tailored parameter. (See Section 3.2.) We based its value on the observation that the largest amount of data that the foreign side will normally want to send is a screenful of data. The goal is to arrange things so that the foreign side can send an entire screenful of data, namely 2000 bytes, without having to wait for a window update which will increase the time it takes to send a screenful of data. To prevent silly window we must update the window in large

increments. If we never update the window by less than 500 bytes, a reasonably large packet size, then the maximum window size needs to be at least 2500 bytes.

Because the personal computer Internet layer does not use tasking for reasons discussed in the previous section, a TCP routine is also the top level procedure in the network process. This routine passes to the signal handler a pointer to the network task control block so that the network task will awaken when a packet comes in from the network. TCP uses this signal as a hint to run and poll Internet for a packet. Internet does not always produce a usable packet; for example if the incoming packet has a bad Internet header checksum, Internet will not return a packet.

The first thing TCP will do when it receives a packet is checksum the packet and make sure that the received checksum agrees with the computed checksum. If the packet acknowledges data, TCP will adjust the output packet. If the incoming packet acknowledges all the outgoing data, then TCP sets the output packet data length to zero and turns off the retransmission timer. If the incoming packet acknowledges only part of the data, then TCP copies the unacknowledged data to the beginning of the packet data area. The amount of data that TCP copies is usually just a few bytes. TCP never has to explicitly remove old data since new data will be copied over the old data, and because TCP passes the size of the packet to Internet which sends only the front of the packet which is new, and not the end of the packet which might be old.

If the incoming packet has data, TCP will check to make sure that it is within its advertised window; otherwise, it is discarded. If it is out of sequence, TCP will put it in the out-of-sequence packet buffer described below. Otherwise, it will awaken the send task to send an acknowledgment, and call Telnet, passing it the new data and any data from the out-of-sequence data buffer that follows the new data in sequence.

To help make sure that the personal computer does not run out of packet buffers, TCP does not acknowledge data or update the window until it has processed all incoming packets in the used queue. Unfortunately, this feature turned out to cause lock step when a foreign

host quickly sent enough data to fill the local TCP window. Because the interrupt driven routines can receive packets faster than the high level protocols can process them, if the foreign host sends a window full of data quickly, then TCP will not acknowledge any of the data or update the window until it has received the entire window full, (assuming that no data is lost.) Meanwhile, the foreign TCP can not send any more data until it receives an updated window advertisement. Thus the connection effectively becomes half-duplex for a short period of time. This problem is not too serious because with our window strategy it only occurs when the foreign host sends more than a screenful of data; however, it did affect some performance tests described in Chapter 6. Section 7.1 discusses some ways of eliminating this problem.

The size of the out-of-sequence data buffer is related to the advertised window size. Because the window is the maximum amount of data that the foreign TCP can have outstanding, it is also the maximum amount of data that might arrive out of sequence. Thus the out-of-sequence buffer must be at least 2500 bytes long, the smallest maximum window size that will always allow a screenful of data to arrive uninterrupted by window updates.

TCP creates two arrays to store out-of-sequence data. One is a character array and one is a sequence number array. TCP stores bytes of data in the character array element indexed by low order bits of the byte's sequence number. TCP stores the byte's sequence number itself in the corresponding element of the sequence number array. The test to check if TCP is storing a particular byte of data in the character array is to see if the sequence number array element indexed by the low order bits of the wanted byte's sequence number is the wanted byte's sequence number itself. If it is, then the wanted byte is in the corresponding character array element.

This scheme has the properties:

1. The test to see if the next byte of data needed by Telnet is buffered, which has to be done every time an incoming packet is received, is quick. It consists of a subroutine call, a logical AND operation, indexing an array, and a comparison of two long numbers.

2. Storing or retrieving a single byte of data is only slightly more complicated. Since most packets in a Telnet connection contain only a single byte of data, most stores and retrieves are fast.

3. Storing or retrieving larger amounts of data is not unacceptably slow.

4. Nothing has to be done to the arrays at any other time.

This scheme is rather space intensive. The character array is 4098 bytes long, the power of two greater than the 2500 bytes needed to prevent delays in sending screenfuls of data. Each sequence number is four bytes long so the sequence number array is 16,394 bytes long. However, speed turned out to be a problem on the IBM Personal Computers while plenty of memory space was available so we felt this trade-off of space for speed was worthwhile.

In practice, incoming packets can take more than a second to process. If the local host has a 4000 byte window advertised and the foreign host sends 4000 bytes in medium sized packets, it might take the local TCP and Telnet a few seconds to process all the incoming information. Lengthy processing can be a problem if the user types a character just after the network task starts processing the first packet. If the network task never gives up control until it processes all incoming packets, then the user's character can be delayed a few seconds. This delay will be especially bad if the foreign host for some reason ignores our window and keeps sending packets preventing the user from typing an abort output or break command.

The solution is that once the TCP procedure in the network task has processed one incoming packet, if another is waiting to be processed, TCP should call Telnet and ask it if the user task needs to run. If so, TCP will leave a message in a variable for the TCP procedure that runs in the send task telling it not to update the window, and then TCP in the network task will block. The user task will run next and process all the characters that the user has typed. The send task will then run, sending the user's characters. The TCP procedure in the send task should not update the window until the network task has finished processing all the incoming packets. Otherwise, the foreign host may send even more packets. When the send task is done, the TCP routine in the network task can continue processing right where it stopped.

5.7 The Heath 19 Terminal Emulator and Other I/O Routines

The IBM BIOS (Basic I/O System) provides interrupt driven routines for handling characters typed on the keyboard. It places codes for the keys hit in a buffer. An Heath 19 terminal emulator function will, when Telnet calls it, process these codes as if the user hit the same keys on an Heath 19 terminal. It returns ASCII characters.

No signal (see Section 5.1) exists for striking the keyboard because writing such code would have required changing BIOS. Telnet polls the character code buffer with the Heath 19 function instead. Telnet schedules the user task to run as often as possible, and each time it runs it polls the character code buffer. The character code buffer has space for fifteen codes which is large enough so that under normal circumstances the user cannot type quickly enough to overflow it.

Another terminal emulator function uses BIOS to put characters on the screen in a manner that simulates an Heath 19 terminal.

5.8 Telnet

The user calls the Telnet program with the name or address of a foreign host as an argument. (Internet sends a packet to a name server to resolve a name.) Telnet first initializes tasking by setting up the user process on the main program stack, and then it calls the TCP initialization routine which sets up the remaining tasks. Next, Telnet calls the TCP function which tries to open a connection to the Telnet server at the specified address. TCP calls one Telnet routine if the connection opens successfully and another if a time-out occurs. When the connection opens, Telnet informs the user and uses the Heath 19 function to check if the user has typed any characters. Because characters typed by the users do not cause interrupt driven code to awaken the user task, Telnet must awaken itself before it blocks in order to be able to run again to check if the user has typed anything.

Telnet provides TCP with a function to call when the connection closes. This does little more than print "closed" on the terminal and return to command level on the personal

computer. Telnet also provides a function that will tell TCP if the user task needs to run to process a character. Finally, it provides TCP with a function to call with Telnet data. This function passes data to the Heath 19 terminal emulator which puts it on the screen, and it also responds to Network Virtual Terminal negotiation requests.

Telnet can negotiate remote echo. It also provides the user with the ability to send a Telnet break command in TCP urgent mode, and the Telnet ***are you there*** command. Telnet allows the user to chose to send data after every character, in which case Telnet calls the TCP push function after every character; or every carriage return, in which case only calls the push function after every carriage return. Finally, Telnet provides the user with a command which prints Telnet connection status information on the screen, and asks TCP to print its status.

Chapter Six

Performance Testing and Evaluation

This chapter describes a series of experiments used to evaluate the performance of the personal computer remote login protocol.

6.1 Some Method Notes

This chapter describes two types of tests: tests between two personal computers connected by an RS-232 line, and tests between a personal computer and a foreign host via the packet concentrator and a network. Tests between two personal computers provided us with a controlled environment in which to conduct performance testing. Of the two types of tests, those between two personal computers had fewer variables affecting performance, and we found it to be the easier environment in which to measure the limits of performance. We used tests across the network to measure performance under conditions of actual use.

One additional set of tests--those between two personal computers connected by the packet concentrator--would also have been useful. These tests would have enabled us to isolate and identify the performance constraints imposed by the packet concentrator. Unfortunately, at the time we tested the protocols, the packet concentrator did not support simultaneous connections to multiple personal computers.

Whenever we needed an arbitrary transfer rate for tests between two IBM Personal Computers, we chose 9600 bits per second.This was originally the fastest rate at which the packet concentrator could run, and we thus thought that it would be the most useful rate in practice. 9600 bits per second is also the fastest rate at which IBM suggests running the serial line.

In the experiments described below, we timed events by finding the difference between the

value of a clock variable at the start of an event and its value at the end of an event. The IBM personal computer provided a clock routine that ran once every 55 milliseconds (at 18.2 Hertz) to which we added a piece of code that updated the clock variable. Unfortunately, this clock was too-coarse grained to accurately measure many interesting events. An example will illustrate this problem. (In this example, when we say that the clock *ticked*, we mean that the clock routine updated the value of the clock variable. We will also refer to the value of the clock variable as the value of the clock.)

Imagine that event A is three-fourths of the clock's period in length. If A occurs within a quarter period after the clock ticks, then the value of the clock after A is over will be the same as it was before A started. Alternatively, if A occurs between one-quarter and one clock periods after the clock ticks, then the value of the clock when A is over will be one period greater than its value just before A started. If the clock and A are independent, then one-fourth of the time, A should occur within one quarter period of a clock tick, while three-fourths of the time, A should occur between one-quarter and four-quarter clock periods after the clock ticks. Thus we have a .25 probability of measuring A as taking zero clock periods, while we have a .75 probability of measuring A as taking one clock period, although neither of these values are particularly close to the true value of A.

We can use two techniques to more accurately measure events that are fast relative to the clock such as event A. With one technique we run fast events many times, one after another, and find the time for a single event by dividing the time for the series by the number of events in the series. An alternative technique is to sum the experimental times of many individual independent events and divide the sum by the number of events to obtain an average time. We used the first technique when possible, because it deterministically bounds the inaccuracy of the experiment due to the clock, while the second only probabilistically bounds the inaccuracy. The second technique was useful when an event was not repeatable, as when the event sometimes occurred unsuccessfully and an unsuccessful event had a value which was uninteresting, but differed greatly from the value of a successful event. In this case the deterministic technique would produce an average value which was skewed by the inclusion of the times for unsuccessful events, while we could selectively ignore unsuccessful events with the probabilistic approach.

6.2 Tests in a Controlled Environment

We performed the experiments in this section with two IBM personal computers connected by an RS-232 wire that was slightly less than a meter in length. These tests isolated the performance of the IBM Personal Computer software and hardware from the effects of an external network.

6.2.1 Detailed Performance Measurements

We designed the first set of experiments to find out how and where the protocols spent time. We did this by sending packets of various sizes from one personal computer to another and back, and then noting the time these transfers took. This time included:

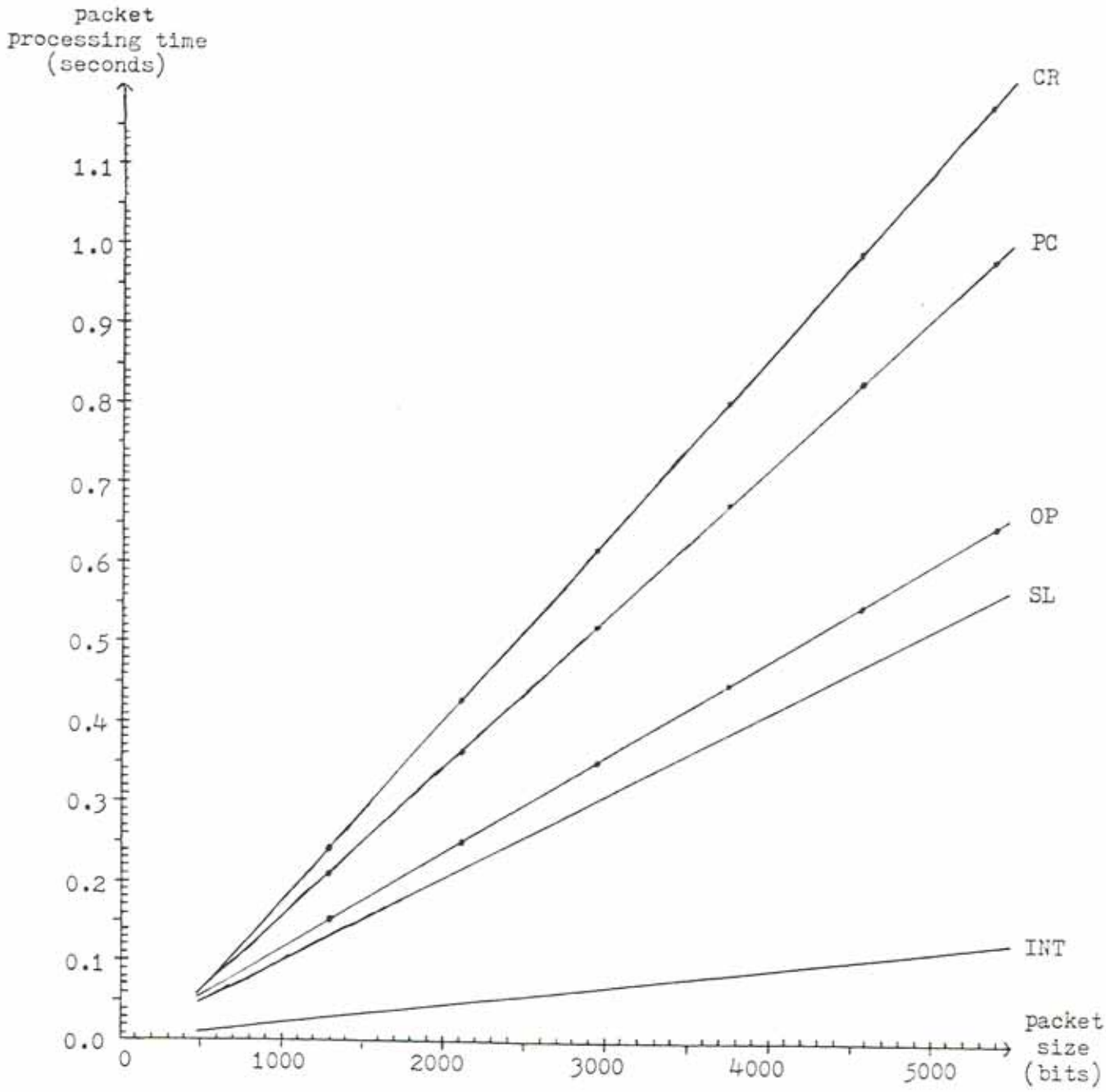
- the time the first personal computer needed to get the packet ready to send,
- the time this packet spent on the serial line,
- the time the second personal computer needed to process the incoming packet,
- the time the second personal computer needed to format a packet containing the same data incoming packet,
- the time this new packet took to get back to the first personal computer,
- and finally the time the first personal computer needed to process this new packet.

In summary, the time included the protocol overhead of sending two packets and receiving two packets, and the time two packets spent on the serial line.

Because the time to do all this was only a few clock periods, the clock was too coarse to measure it accurately. Thus we ran one hundred of these operations one after another, and divided the resultant total time by one hundred to get the time for a single transfer. Because we sent the same data one hundred times, the protocols needed to copy the data into an output packet only once in each direction.

Figure 6-1 presents the results of such timing measurements. It shows the packet size in bits

Figure 6-1:Packet Transfer Times Between Two Personal Computers



CR = packet contained one carriage return for every eighty printable characters

PC = packet contained only printable characters and no carriage returns

OP = packet data was not printed on the screen

SL = time packet was on the serial line

INT = interrupt driven code overhead to receive the packet

Note: The serial line speed was 9600 bits per second.

plotted along the horizontal axis and the transfer time plotted on the vertical axis. Instead of plotting the size of two packets versus the time to send two packets, the time to receive two packets, and the time two packets were on the net, we divided the number of bits and the transfer time by two to get the time spent processing a single packet. Each point in Figure 6-1 is the average of three timings. With these points, we used linear regressions to calculate the lines in Figure 6-1.

In addition to data, each packet had twenty bytes of TCP header, twenty bytes of Internet header, four bytes of local net header, a READY byte, and ACK byte and an END byte. The minimum size packet, the size of a packet with no data, was 47 bytes long. On the personal computer, each byte was eight bits long, but LLP added a start and stop bit to each byte sent over the serial line. Thus sending a 47 byte packet meant sending 470 bits.

Line CR of Figure 6-1 shows the results where the packet data contained one carriage return after every eighty printable characters. Carriage returns turned out to be much more expensive to process than printable characters because they involved scrolling lines on the screen.

Line PC of Figure 6-1 shows the results of transfers similar to the first set, except that we replaced each carriage return with two printable characters. (To distinguish a carriage return that acts as a newline from a carriage return that is really a carriage return, Telnet represents the former as a carriage return followed by a linefeed and the latter as a carriage return followed by a null.)

A third set of transfers dispensed with printing characters on the screen altogether. This set was similar to the second set except that we patched out the call to the terminal emulator routine that wrote characters on the screen. Line OP in Figure 6-1 shows he results of these transfers. (OP stands for Other Protocol overhead.)

Line SL in Figure 6-1 shows the amount of time that a packet of a particular size had to spend on the serial line at a transfer rate of 9600 bits per second. (We calculated line SL rather than measured it.) This line represents the theoretical minimum amount of time that the protocol needed to transfer a packet using the serial line.

The interrupt driven code that handled characters on the serial line ran faster than 9600 bits per second. Concurrent with the time that characters were on the line but interrupt driven code was not running, the processor ran non-interrupt driven code, (the code that the interrupt driven routines preempted). For outgoing packets, this code was just a wait loop in the non-interrupt driven part of the low level protocol. (See Section 5.3.) For incoming packets, any code could run.For example, while the network sent one packet to the personal computer, the high level protocols could process the previous packet, although interrupt driven routines would preempt them for a short while each time a new byte came in from the serial line. The fact that interrupt driven routines and high level protocols ran concurrently (interspersed) was important in explaining performance figures presented later in this chapter.

Because the interrupt driven routines that handled the net ran interspersed with high level code, and because they turned off interrupts including the clock interrupt, we had no convenient way of directly measuring the amount of time that they used. We indirectly calculated a conservative estimate of this time by looking up in the 8088 User's Manual, the number of clock cycles that the interrupt level code that handled the average character used, and adding 10 percent to this number. (Special characters such as RDY took much more time to process than the average character. No characters took less time to process. The 8088 User's Manual states, "With typical instruction mixes, the time actually required to execute a sequence of instructions will typically be within 5-10% of the sum of the individual timings... Cases can be constructed, however, in which execution time may be much higher..." [12].) This estimate suggested that the interrupt level routines took 231 microseconds to process each byte that they received from the serial line. We plotted this estimate on Figure 6-1 as line INT.

From Figure 6-1 we calculated a number of useful statistics. (The formulas that we used to calculate them are summarized in Part A of Table 6-1.) The difference between line CR and line PC is the carriage return overhead (CRO). This was the amount of additional time that the protocols spent processing carriage returns compared with processing printable characters. The carriage return overhead depended upon the frequency with which carriage

Table 6-1: Breakdown of Packet Transfer Times Between Two Personal Computers

A. Formula Used to Calculate Detailed Times

```
Carriage Return Overhead.....................CRO = CR (size) - PC (size)
Printing Character Overhead..................PRO = PC (size) - OP (size)
Other Overhead--Fixed........................OOF = OP (size = 0) - SL (size =0)
Other Overhead--Variable.....................OOV = OP (size) - OOF - SL (size)
Available Concurrent Processing Time.......ACT = SL (size) - INT (size)
```

B. Detailed Times

statistic	values for various packet data sizes				per byte value
	0 bytes (sec.)	1 byte (sec.)	466 bytes (sec.)	500 bytes (sec.)	(usec/byte)
CRO	0.0000	0.0004	0.1825	0.1959	392
PCO	-0.0001	0.0006	0.3225	0.3460	692
OOF	0.0100				
OOV	0.0000	0.0002	0.0357	0.0768	154
SL(size)*	0.0490	0.0500	0.5344	0.5698	1042
INT(size)**	0.0109	0.0111	0.1185	0.1264	231
ACT	0.0381	0.0389	0.4159	0.4434	811

Notes: * This statistic was not measured experimentally. It was calculated by dividing the total packet size in bits by 9600 bits/second .

** This statistic was based on the estimated amount of time spent in interrupt driven code. This estimate may be very inaccurate.

returns appeared in the data steam. In these experiments, we sent one carriage return for every full line of printable characters, i.e., one for every 80 printable characters. Table 6-1 Part B shows the carriage return overhead for packets of various sizes. If for some packet size, we divided the carriage return overhead by the number of data characters transferred, we found that the cost per data character of having carriage returns as opposed to having printable characters was 392 microseconds. Since packets contained one carriage return in every eighty-two bytes of data, the processing cost per carriage return was 32 milliseconds.

This is apparently the time required to move the previous screen contents up one byte (scrolling).

The difference between line PC and line OP of Figure 6-1 is the amount of time that the software spent writing printable characters on the screen. This statistic also appears in Table 6-1.

By subtracting line OP from line SL, we found the protocol overhead of the non-interrupt driven routines that was not associated with printing things on the screen. We separated this other protocol overhead into two components: a fixed overhead which was irrespective of the amount of data that the packet contains and included a lot of header processing, and a variable overhead which depended on the amount of packet data. The fixed overhead was the difference between the two lines when the packet contained no data (470 bits). The variable overhead was the difference between the lines OP and SL at other packet sizes less the fixed overhead. The fixed overhead was 0.01 seconds, while the variable overhead was 154 microseconds per byte of data.

A problem with these measures of other protocol overhead was that they included the overhead of both sending and receiving packets. We had no convenient way of determining how much of this overhead the code that sent packets caused, and how much the code that received packets caused. We attributed half the overhead to each. Attributing half the cost to each probably underestimated the amount of time necessary to receive packets, but for most of the calculations in this chapter, a low estimate of the time needed to receive a packet was a conservative estimate.

Another interesting statistic was the amount of time available to non-interrupt driven routines when a packet came in from the serial line. This statistic measured the time available for concurrent processing in the protocols, (as explained on page 55). Because of available concurrent processing time, we did not expect the protocols to take as long to send ten packets from one personal computer to another and back, as we did to send one packet back and forth ten times. In the former case the protocols processed and received packets

concurrently, while in the latter case the protocols completely processed one incoming packet and sent a packet before they started to receive the next packet. In the latter case the protocols wasted the available concurrent processing time. The available concurrent processing time is equal to the difference between the line SL, the amount of time the packet was on the serial line, and line INT, the amount of time used by the interrupt driven code that processes incoming characters, and it appears in Table 6-1.

6.2.2 The Maximum Data Transfer Test

We based a number of the statistics in Figure 6-1 and Table 6-1 on estimates as well as measurements, and we wished to double check the accuracy of these estimates. We devised an experiment in which one IBM Personal Computer sent large packets at a second as quickly as possible for ten minutes. Because the first personal computer could send packets faster than its counterpart could receive them, (the sender did not have to print characters on the screen), the first personal computer kept the second one busy for the duration of the test. We counted the number of data packets that the second personal computer received, and the number of packets acknowledging data that it sent. Using these two numbers and the information provided by Figure 6-1 and Table 6-1, we predicted the amount of time that a personal computer needed to receive and send these particular numbers of packets. The closer this predicted time came to ten minutes, the actual transfer time, the greater the accuracy of Figure 6-1 and Table 6-1.

The personal computer that received data originally offered a window of 4000 bytes. The sending personal computer filled this window with eight 500 byte packets, and waited for a window update before sending more packets. As the first packet of these eight packets arrived, Telnet, Internet and TCP had nothing to do concurrently with the interrupt driven code that received characters from the serial line. When the second and succeeding six packets arrived, the higher level protocols had data to process concurrently with the interrupt driven routines. Because the receiving personal computer did not update the window or acknowledge data until it processed all the packets that had arrived, (see Section 5.6), and because the sender could send packets faster than the receiver could process them,

Table 6-2: Predicted Time of the Maximum Data Transfer Test

```
Packets Sent:    92              Packets Received:   734
 91 full groups of eight packets and 1 partial group of 6 packets

First Packet Received in Each Group of Eight:
  Carriage Return Overhead            =      CRO (500)   =  0.1959  seconds
  Printable Character Overhead        =      PCO (500)   =  0.3460  seconds
  Other Protocol Overhead - Fixed
    for incoming packet               =  1/2 OOF         =  0.0050  seconds
  Other Protocol Overhead - Variable
    for incoming packet               =  1/2 OOV (500)   =  0.0384  seconds
  Interrupt Driven Code Overhead
    for incoming packet               =      INT (500)   =  0.1264  seconds
  Available Concurrent Proc. Time     =      ACT (500)   =  0.4434  seconds
  ------------------------------------------------------------------------
  Total                                                   =  1.1551  seconds

Next Seven Packets Received in Each Group of Eight:
  Carriage Return Overhead            =      CRO (500)   =  0.1959  seconds
  Printable Character Overhead        =      PCO (500)   =  0.3460  seconds
  Other Protocol Overhead - Fixed
    for incoming packet               =  1/2 OOF         =  0.0050  seconds
  Other Protocol Overhead - Variable
    for incoming packet               =  1/2 OOV (500)   =  0.0384  seconds
  Interrupt Driven Code Overhead
    for incoming packet               =      INT (500)   =  0.1264  seconds
  ------------------------------------------------------------------------
  Total                                                   =  0.7117  seconds

Packet Acknowledging Data:
  Other protocol Overhead - Fixed
    for outgoing packet               =  1/2 OOF         =  0.0050  seconds
  Time Packet was on the Serial Line  =       SL (0)     =  0.0490  seconds
  ------------------------------------------------------------------------
  Total                                                   =  0.0540  seconds

Time for the Sending Personal Computer to Process the Acknowledgment
     and Start Sending New Data:
  Other Protocol Overhead - Fixed
    for incoming ACK packet           =  1/2 OOF         =  0.0050  seconds
  Other Protocol Overhead - Fixed
    for outgoing data packet          =  1/2 OOF         =  0.0050  seconds
  Other Protocol Overhead - Variable
    for outgoing data packet          =  1/2 OOV (500)   =  0.0384  seconds
  ------------------------------------------------------------------------
  Total                                                   =  0.0484  seconds


    92        First Packets       =  106.3  seconds
(734 - 92)    Later packets       =  456.9  seconds
    91        Ack. Packets        =    4.9  seconds
    91        Foreign Overheads   =    4.4  seconds
-------------------------------------------------------------------------
Total                             =  572.5  seconds  =  95% of 600 seconds
```

the receiver only sent one acknowledgment for every eight packets sent. The packet that contained the acknowledgment also offered the sender a new 4000 byte window, and the process started over again. The receiving personal computer received 734 packets and sent 92 packets. We would actually only have predicted that it would send 91 packets, one for each complete group of eight packets received. We ignored the extra packet in our figures.

Table 6-2 shows the results of this test. We accounted for 572.5 seconds out of the actual 600 second transfer time (95%), which shows that our estimates in Figure 6-1 and Table 6-1 were fairly accurate. This test also shows the maximum rate at which the personal computer protocols could handle large amounts of data on a 9600 bit per second line. In ten minutes the personal computers transferred 734 data packets, each of which contained 5470 bits of which 4000 were data bits (500 data bytes) for a total transfer rate of 6770 bits per second and an effective data transfer rate of 4900 bits per second.

6.2.3 Performance Limitation Predictions

We used Figure 6-1 and Table 6-1 to predict the minimum amount of time that the protocol software needed to handle a screenful of data. The screen held 24 eighty character lines separated by 23 carriage returns. A screenful of data required 1966 data bytes which we assumed would be sent in three packets with 500 data bytes, and one packet with 466. We further assumed that the serial line speed was perfectly adjusted to the software speed so that there was no available concurrent processing time, i.e., that lines SL and INT in Figure 6.1 overlapped. From these assumptions we concluded that the protocol software needed about 2.8 seconds to receive a screenful of data. (Table 6-3 gives the details of this calculation.) This implied that for a screenful of data, the protocols have a maximum total transfer rate (data and header) of 7700 bits per second, and a maximum effective data transfer rate of 5600 bits per second.

We can also break down the overhead for a screenful of data into its components. When we do this we find that the speed at which the terminal emulator routines write data on the screen is the factor which most limits the handling time for a screenful of data. They need

Table 6-3: Minimum Time Needed to Handle a Screenful of Data

```
1966 characters/screenful (1920 printable and 23 carriage returns)
3 packets of 500 bytes, 1 packet of 466 bytes.


Minimum Overhead for Incoming Packets with 500 Data Bytes:
  Carriage Return Overhead            =      CRO (500)   =  0.1959  seconds
  Printable Character Overhead        =      PCO (500)   =  0.3460  seconds
  Other Protocol Overhead - Fixed     =  1/2 OOF         =  0.0050  seconds
  Other Protocol Overhead - Variable  =  1/2 OOV (500)   =  0.0384  seconds
  Interrupt Driven Code Overhead      =      INT (500)   =  0.1264  seconds
  ------------------------------------------------------------------------
  Total                                                  =  0.7117  seconds

Minimum Overhead for Incoming Packets with 466 Data Bytes:
  Carriage Return Overhead            =      CRO (466)   =  0.1825  seconds
  Printable Character Overhead        =      PCO (466)   =  0.3225  seconds
  Other Protocol Overhead - Fixed     =  1/2 OOF         =  0.0050  seconds
  Other Protocol Overhead - Variable  =  1/2 OOV (466)   =  0.0357  seconds
  Interrupt Driven Code Overhead      =      INT (466)   =  0.1185  seconds
  ------------------------------------------------------------------------
  Total                                                  =  0.6642  seconds

3 Packets with 500 Data Bytes  =  2.1351  seconds
1 Packet  with 466 Data Bytes  =   .6642  seconds
-----------------------------------------------
Total                          =  2.7993  seconds
```

2.13 seconds to write 1966 bytes on the screen. (This figure is the carriage return overhead and printable character overhead of three 500 data byte packets and one 466 data byte packet. We double checked this figure with a simple program that printed a 1966 byte character array on the screen one element at a time.) Thus no matter how fast we make the rest of the code, the I/O routines will prevent the remote login protocol from handling a screenful of data at a speed faster than 7400 bits per second.

The terminal emulator and I/O code for the personal computer is well written. Improving them enough to make a significant change in the handling time for a screenful of data would be difficult. We should mostly attribute the 7400 bps figure to the limitations of the of the personal computer's microprocessor.

Another interesting statistic is the maximum transfer rate that we would expect if we

replaced the serial line and interrupt driven code with a faster network connection. We can find an upper bound on this figure by adding the carriage return overhead and printable character overhead to half of the fixed and variable protocol overheads for three 500 data byte packets and one 466 data byte packet. The total overhead comes to 2.30 seconds, implying a maximum transfer rate of about 6800 bits per second.

From our estimates of the speed at which the interrupt driven routines that fetch bytes from the serial line run, we can predict the maximum rate at which the interrupt driven routines can receive packets. Since the average byte takes 231 microseconds to process, we expect the interrupt driven routines to drop bytes if the bytes came in over the serial line at a rate faster than 43,000 bits per second. In fact, we expect the interrupt driven routines to drop bytes at rates substantially below this because some bytes take longer to process than 231 microseconds. In practice the interrupt driven routines did not receive any packets correctly at 43,000 bps, and they received only half of the packets correctly at 38,400 bits per second, a speed which provides 260 microseconds to process each byte. They did not correctly receive 3% of the packets at 28,800 bits per second, which supported the conclusion that a few bytes took longer than 347 microseconds to process. (Incorrectly received packets are packets whose checksum field does not match the computed checksum.)

6.2.4 Character Echoing Time

Section 3.1 showed that the amount of time a remote login implementation takes to echo a character is an important measure of its performance. Between two IBM personal computers, the echoing time is the amount of time necessary for a packet with a single data byte to go from one personal computer to another, and back to the first. The character is printed on the screens of both personal computers. Table 6-4 shows the character echoing time at various serial line speeds.

The columns of Table 6-4 labeled "Predicted Times" show predicted character echo times derived from Figure 6-1 while the columns marked "Experimental Times" show the results of a special test in which 1000 character echoes were run one after another, and the result

Table 6-4: Character Echoing Time between Two Personal Computers

```
Overhead Not Dependent on Line Speed - One Way
     (One Packet with a Single Data Character):
  Printable Character Overhead          =  PCO (1)   =  0.0006 seconds
  Other Protocol Overhead - Fixed
    both incoming and outgoing          =  OOF       =  0.0100 seconds
  Other Protocol Overhead - Variable
     both incoming and outgoing         =  OOV (1)   =  0.0002 seconds
  ------------------------------------------------------------------
  Total                                                =  0.0108 seconds


Overhead Not Dependent on Line Speed - Round Trip:
                     =  2 * One Way Overhead         =  0.0216 seconds


One Packet with a Single Data Character has 480 bits.

One Way Line Time      =  480 bits / Line Speed
Round Trip Line Time   =  2 * One Way Line Time
Total Round Trip Time  =  Round Trip Line Time
                             + Round Trip Overhead Not Dependent on Line Speed
```

	Predicted Times		Experimental Times	
Line Speed (bps)	Line Time Round Trip (seconds)	Total Time Round Trip (seconds)	Total Time Round Trip (seconds)	Total Time Minus Predicted Line Time (seconds)
4,800	0.2000	0.2216	0.2222	0.0222
9,600	0.1000	0.1216	0.1219	0.0219
19,200	0.0500	0.0716	0.0718	0.0218
23,040	0.0416	0.0632	0.0638	0.0222

was divided by 1.000. Notice that the predicted and experimental values in the columns labeled "Total Time Round Trip" are quite close.

Table 6-4 shows that most of the character echoing time is due to serial line overhead. The protocols spend only about 20 milliseconds processing the packets exclusive of processing related to the serial line. If we could replace the serial line with a faster network connection we might improve performance by up to a factor of three compared with the current performance at a serial line speed of 23.040 bits per second.

6.2.5 Handling Time for a Screenful of Data

Section 3.1 also mentioned the importance of a screenful of data to a Telnet connection. Table 6-5 shows predicted and experimental values of the handling time for a screenful of data transferred between two personal computers. The times shown in Table 6-5 measure from the instant just before the protocols print the first character of the screenful on the screen, to the instant just after they print the last character on the screen. These measurements do not include the time the first packet in the screenful spent on the serial line, or the time the protocols spent processing the first packet before they printed the first character on the screen. We chose to measure the handling time for a screenful of data in this way because we were interested in measuring the performance of the personal computer protocol code rather than the performance of the foreign hosts or the performance of the interconnecting networks.

To predict the handling time for a screenful of data used a different method at serial line speeds of 9600 bits per second and greater, from that used at 4800 bits per second. Section 6.2.3 showed that at speeds greater than 7700 bps, the software overhead limits performance. At these speeds, the protocol software does not have to wait for data because the available concurrent processing time is less time than the protocols need to process the incoming data. We calculated the predicted overhead at these speeds from the amount of time the protocols need to process the packets containing a screenful of data. At a serial line speed of 4800 bps, the time that the protocol software takes to process a packet is less than the time that the packet is on the serial line, so some available concurrent processing time goes unused, and the time a packet is on the serial line affects performance. At 4800 bps, the screenful handling time is found by adding the time that the packets are on the serial line to the time needed to process the last packet. (The protocols process each of the other packets while its successor is on the serial line.)

The difference between the predicted and experimental values of Table 6-5 can be only partly explained by clock inaccuracy. At speeds of 9600 bps and above, the two sets of values differ by between two and three clock periods. We appear to have overlooked some factor in calculating the predicted values. The large difference between the predicted and

Table 6-5: Time Needed to Handle a Screenful of Data Between Two PCs

```
A. Predicted Overhead at Speeds Where Protocol Overhead is the Bottleneck

Time spent processing first packet after first character gets
     printed:
  Carriage Return Overhead            =      CRO (500)  =  0.1959  seconds
  Printable Character Overhead        =      PCO (500)  =  0.3460  seconds
  Other Protocol Overhead - Variable  =  1/2 OOV (500)  =  0.0384  seconds
  ------------------------------------------------------------------------
  Total                                                  =  0.5803  seconds

Overhead not Dependent on Line Speed for Remaining Incoming
     Packets with 500 Data Bytes:
  (See Table 6.3)                                        =  0.7117  seconds

Overhead not Dependent on Line Speed for Final Packet With
     466 Data Bytes:
  (See Table 6.3)                                        =  0.6642  seconds

Total Overhead at Speeds Where Protocol Overhead is the Bottleneck
  1  first 500 data byte packet              =  0.5803  seconds
  2  remaining packets with 500 data bytes   =  1.4234  seconds
  1  final packet with 466 data bytes        =  0.6642  seconds
-------------------------------------------------------------
  Total                                      =  2.6679  seconds


B. Predicted Overhead at Speeds Where the Serial Line is the Bottleneck

Total Time Spent on the Serial Line  =  Packet Size / Line Speed
  500 data byte packet (5470 total bits) at 4800 bps    =  1.1396  seconds
  466 data byte packet (5130 total bits) at 4800 bps    =  1.0688  seconds

Time to process a 466 data byte packet exclusive of Serial Line Time:
  Carriage Return Overhead            =      CRO (466)  =  0.1825  seconds
  Printable Character Overhead        =      PCO (466)  =  0.3225  seconds
  Other Protocol Overhead - Fixed     =  1/2 OOF        =  0.0050  seconds
  Other Protocol Overhead - Variable  =  1/2 OOV (466)  =  0.0357  seconds
  ------------------------------------------------------------------------
  Total                                                 =  0.5457  seconds


Total Overhead at 4800 bits per second:
  Time second packet is on the serial line  =  1.1396  seconds
  Time third packet is on the serial line   =  1.1396  seconds
  Time final packet is on the serial line   =  1.0688  seconds
  Time to process the final packet          =  0.5457  seconds
-------------------------------------------------------------
  Total                                     =  3.8937  seconds
```

Table 6-5 (Cont.)

C. Comparison of Predicted and Experimental Values

Line Speed (bits per second)	Predicted Value (seconds)	Experimental Value (seconds)
4,800	3.89	4.62
9,600	2.67	2.82
19,200	2.67	2.83
23,040	2.67	2.84

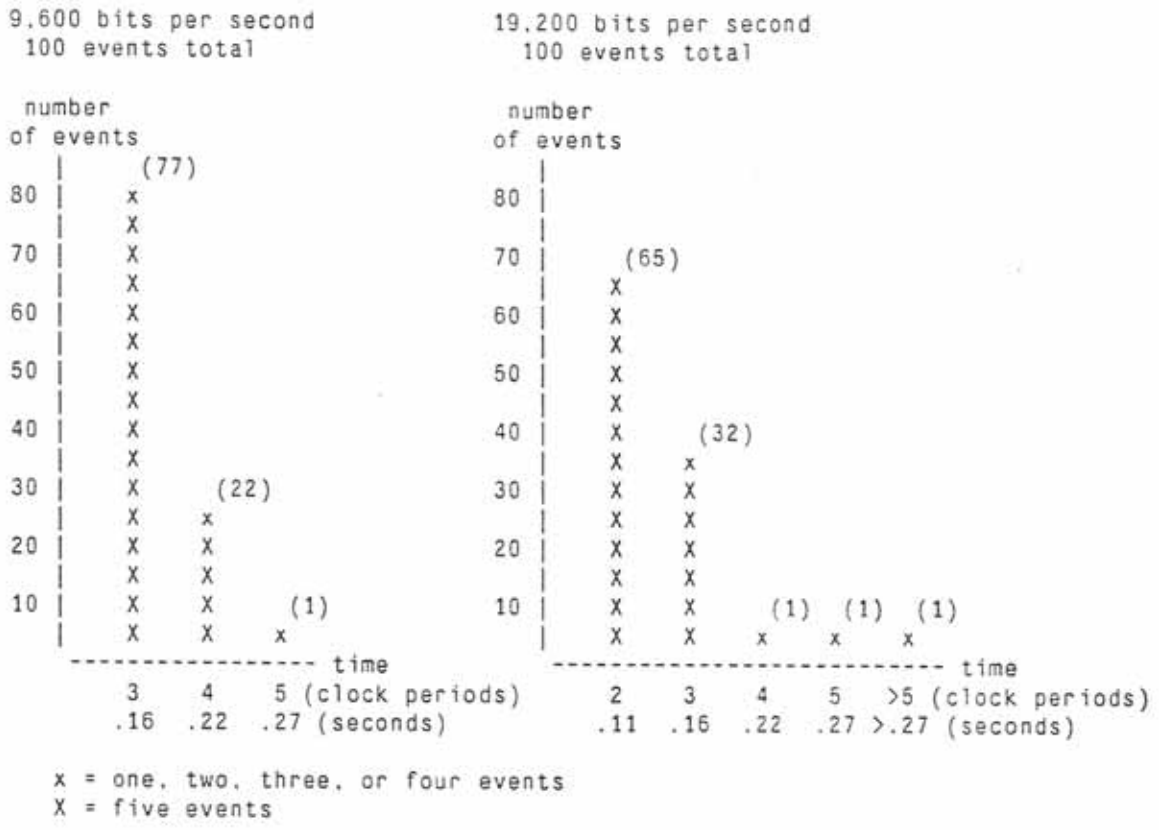
the experimental value at 4800 bps has an alternative explanation. At this speed, the low level protocol had some bugs that caused the receiving personal computer to be unsuccessful in sending packets that acknowledged data. Typically, time-outs would occur in four out of five tests at this speed, and while Table 6-5 shows the results of tests that did not time out, they probably came close to doing so.

6.3 Tests Across a Network

The tests in this section illustrate the performance of the remote login protocol while it communicated with machines on various local networks. These are the machines with which the protocol will be used in practice.

A characteristic of network communication is that its performance can fluctuate significantly from one moment to the next. A network may experience a particularly heavy load in one instant, thereby delaying or losing packets, and work perfectly well the next instant. A foreign host may process a request for information slowly when three of its users simultaneously try to compile large programs, but will respond much more quickly when they finish. The statistics in this section will tend to show best case rather than average case performance, because we wish to illustrate how well the personal computer protocols can work rather than how poorly the network or foreign hosts can respond. We generally used the averaging technique with probabilistic accuracy rather than the technique with deterministic accuracy (see Section 6.1) because this approach allowed us to ignore non-optimal data.

Figure 6-2:Character Echoing Performance with CSR



6.3.1 CSR: A Sample Foreign Host

At this point we shall describe measurements on a connection with one particular foreign host, CSR, to illustrate the sorts of problems that we faced in interpreting data from connections with foreign hosts. CSR is a PDP-11/45 with a Unix operating system that belongs to what was once the Computer Systems Research Group. CSR attaches to a 10 Mbit ringnet which in a gateway in turn connects to the 1 Mbit ringnet of which the packet concentrator is a member.

Figure 6-2 shows the results of 100 individually timed character echo tests on a connection between an IBM Personal Computer and CSR. We used the events that took three or four

clock periods at 9600 bps or two or three periods at 19,200 bps to calculate the best case times. We assumed that the other events non-optimal conditions delayed the other events, and we ignored them.

Figure 6-3:Handling Time for a Screenful of Data from CSR

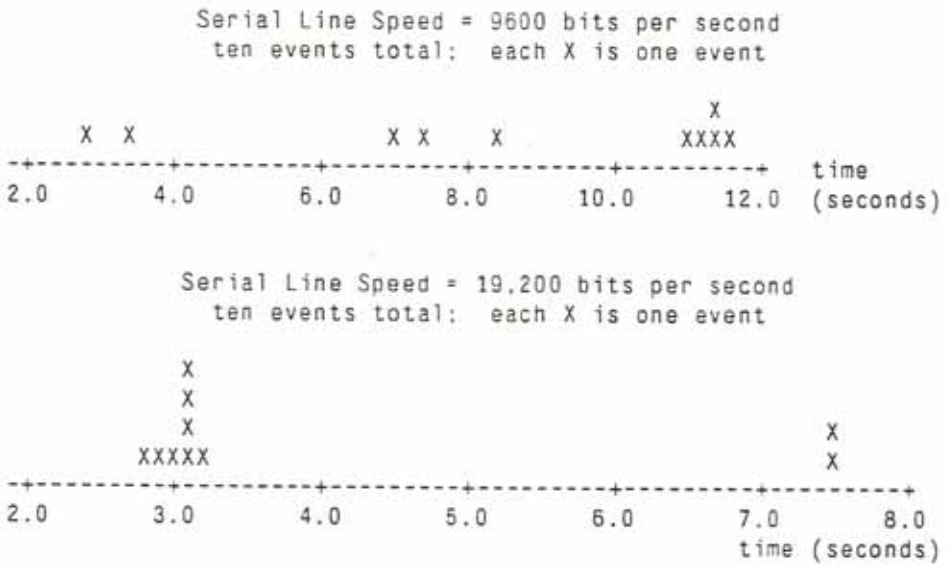


Figure 6-3 shows the results of ten individually timed tests for measuring the handling time of a screenful of data. The number of retransmissions that were necessary to get the data from CSR to the personal computer explains the clumping within the data. Tests which took about three seconds experienced no retransmissions, test which took about seven seconds experienced one retransmission, and tests which took about eleven seconds experienced two retransmissions. Various loads on the network and CSR probably caused the differences between tests within clumps. We found the best case times for this type of test by taking the event with the lowest time at each serial line speed.

The large number of retransmissions was due to a bug in the packet concentrator. Whenever multiple packets came into the packet concentrator in quick succession from a foreign host, the packet concentrator often dropped packets other than the first. Increasing the number of packets in the succession or decreasing the serial line speed increased the chances that the concentrator would drop a packet. This was not a problem for the character echoing test, since it sent packets one at a time.

Table 6-6: Character Echoing Performance With Various Foreign Hosts

Foreign Host	9600 bps	19,200 bps	difference
IBM PC *	0.122 sec.	0.072 sec.	0.050 sec.
CSR	0.175 sec.	0.128 sec.	0.047 sec.
Comet	0.196 sec.	0.136 sec.	0.060 sec.
XX	0.212 sec	0.161 sec.	0.051 sec.
Multics	0.382 sec	0.325 sec	0.057 sec.

* = The packet concentrator was not used between the two IBM Personal Computers.

6.3.2 Character Echoing Performance with Various Foreign Hosts

Table 6-6 shows the character echoing performance with various foreign hosts.[7] Significantly, the echo time with many hosts was under 0.25 seconds, and in one case was under 0.15 seconds, a level of performance that most users would find reasonable. We included a connection between two IBM Personal Computers for reference purposes.

Although we did not expect to be able to predict the exact character echo times for the foreign hosts, we did expect that for the same foreign host, the difference between the times of tests at different serial line rates would be approximately constant. The amount of time saved by speeding up the serial line interface should be independent of the foreign host. Table 6-6 confirmed this expectation.

[7] Comet is a Digital VAX/750 running Unix. XX is a Digital DEC System-20 running Tops-20. Multics is a Honeywell Information Systems 68/80 running Multics, and which we accessed via the ARPANET.

Table 6-7: Handling Time for a Screenful of Data from Various Foreign Hosts

Foreign Host	predicted	9600 bps	19,200 bps
IBM PC *	2.67 sec.	2.82 sec.	2.83 sec.
CSR	2.67 sec.	2.80 sec.	2.91 sec.
XX	2.77 sec.		3.19 sec

* = The packet concentrator was not used between the two IBM Personal Computers.

6.3.3 Handling Time for a Screenful of Data From Various Foreign Hosts

Table 6-7 shows predicted and actual handling times for a screenful of data from various foreign hosts. We assumed that the foreign hosts could send packets faster than the personal computer could process incoming packets, i.e., that the foreign hosts could send packets at a rate faster than 7700 bits per second (see Section 6.2.3), and expected that the personal computer software overhead would limit screenful handling performance just as was the case with transfers between two personal computers. We thus used the same figures to predict performance between a personal computer and a foreign hosts that we used to predict performance between two personal computers.

CSR sent a screenful of data in four packets, exactly the same number of packets as one personal computer sent to the other in the tests between two personal computers. Thus we expected the handling time for a screenful of data between CSR and a personal computer to be very close to the handling time between two personal computers. Table 6-7 confirmed this expectation.

Foreign hosts other than CSR sent each screenful of data in more than four packets. XX and Multics sent a screenful of data in about eight packets, while Comet used more than twenty. These large numbers of packets reacted unfavorably with the bug in the packet concentrator, and prevented us from receiving a screenful of data from most of these hosts

without retransmissions. The only additional host from which we could obtain figures was XX when the serial line speed was 19,200 bps. XX sent a screenful of data in eight packets. We therefore expected a personal computer to take somewhat longer to handle a screenful of data from XX than from CSR or another personal computer. Table 6-7 also confirmed this expectation.

The number of packets that a host uses to send a screenful of data is important for reasons other than its interactions with the packet concentrator bug. Because foreign hosts send data to the packet concentrator faster than the packet concentrator sends packets to the personal computer. the packet concentrator may run out of space and drop packets if the foreign hosts sends the packet concentrator too many packets. Dropping packets would force the foreign host to retransmit the packets which would slow down communication. This problem is not just idle speculation. Comet uses more than twenty packets to send a screenful of data. Because Comet sends these packets to the packet concentrator much faster than the packet concentrator sends them to the personal computer. packets back up at the packet concentrator. Twenty packets is a large enough number to cause the packet concentrator to run out of packet buffers. and throw away some packets in the screenful thus forcing Comet to retransmit the packets. (even if it did not have to retransmit them due to the bug in the packet concentrator.)

This problem of dropped packets highlights two fundamental problem with the TCP. First. the TCP flow control mechanism allows implementations to specify only roughly the maximum amount of information that they can handle. A local TCP implementation specifies its window in data bytes. but if the foreign host sends data in small packets. then the amount of data that the local low level routines must handle may be significantly greater than the amount advertised by the local TCP window. For instance. if a foreign host sent each byte of TCP data to the personal computer in a separate packet. the low level protocol on the personal computer will handle forty-eight bytes for every byte of TCP data. Fortunately. reasonably intelligent TCP implementations that try to send data in packets that are as large as possible can minimize this problem.

A second problem with TCP is that it provides no mechanism for congestion control. Even if the machines on both ends of a connection can keep up with the flow of data, the flow may still overtax some intermediate machine, and cause it to perform poorly. Comet overtaxes the packet concentrator in exactly this way. The link between Comet and the packet concentrator is much faster than the serial line between the packet concentrator and the personal computer. The packet concentrator can not get rid of packets as fast as it receives them, and it sometime runs out of packet buffers and drops packets.

If we make the personal computer's window sufficiently small, the packet concentrator will not run out of buffers. Comet sends packets with about eighty bytes of data. If the personal computer TCP advertises a 240 byte window, then Comet will only send four packets at once, just like CSR. Unfortunately, a 240 byte window implies that the window must be updated eight or nine times for the personal computer to receive a screenful of data. Eight or nine window updates increase the time needed to receive a screenful of data, although not by as much as waiting for retransmission timeouts. Making the TCP window size dependent on the foreign host is very unmodular. Why should the window size of the personal computer TCP depend on the number of buffers in the packet concentrator?

Congestion control is a well known and difficult problem in the field of internetwork communication [13] [14]. A way to prevent congestion at the junction of low speed and high speed networks is of particular importance to us and is a topic for future research.

6.3.4 Memory Size and Code Length

In the Introduction of this paper, we suggested that the limited memory size of desktop personal computers might affect the performance of an internet remote login protocol. In fact, the memory size did not affect performance. The memory of the IBM Personal Computer was sufficiently large that we never had to worry about space in the implementation. In some cases our implementation was actually space intensive. Our buffering strategy for out-of-sequence TCP data is an example.

Table 6-8 shows the code length of various modules used in the personal computer

Table 6-8: Code Length of Various Modules

Module	Code Length (bytes)
Telnet	= 3,156
TCP	= 5,396
Internet	= 1,112
LLP and Packet Manager	= 4,868
Tasking	= 432
C Standard I/O Library and Heath 19 Terminal Emulator	= 9,068
C Runtime System*	= 282
Operating System Area	= 256
Total	= 24,470

Note: * The C runtime system starts C programs running on the IBM Personal Computer and initializes the Standard I/O Library.

protocols. The code for the entire remote login implementation occupied under 25 KB of memory. The "Global and Static Variables" portion of the data segment (see Figure 5-2) was 49.584 bytes long. Of this, the TCP out-of-sequence data buffers took up 20.480 bytes, and thirty-two 704 byte packet buffers used 22.528 bytes of space. Task control blocks, task stacks, and the main program stack used additional memory. Thus the entire remote login package needed about 80 KB of memory, and it could have fit in a 96 KB or 128 KB system.

Chapter Seven

Conclusion

7.1 Suggestions for Improvement

The first improvement we would make to the implementation described by this thesis would be to rewrite Internet so that it used tasking and ran as the top level procedure in the network task. Such a modification would make it easy for Internet to demultiplex packets among various client protocols and to handle fragmented packets and Internet options. With the current system, if two protocols ran on top of Internet, both would have to poll Internet on an incoming packet signal. Only one client protocol would actually get a packet, or perhaps none would if the incoming packet were a fragment. Unsuccessful polling wastes time. If Internet were at the top of the network task, it could call the appropriate client protocol, and all its other clients would not have to run. If only a fragment came in, Internet could put it into a fragment buffer and would not call a client at all.

If Internet could tell TCP the number of packets that are waiting in the used queue for the network task to process them, TCP could update the window when the the protocols had one incoming packet left to process rather than zero packets. Updating the window earlier would allow the foreign TCP to send more packets before the personal computer protocol actually ran out of packets to process. This would increase the amount of concurrent processing within the remote login implementation. It would also solve the TCP lock step problem described in Section 5.6.

The tasking package also has room for improvement. To block, a procedure must call a blocking routine in the tasking package in order to give up control and allow a new process to run. We wrote this blocking routine in C, and it is five lines long. It calls a routine written in assembly language to actually swap the old and new stack pointers. Swapping tasks thus requires two procedure calls. If we write the blocking routine entirely in assembly language,

the code to swap tasks could be combined into a single procedure, a process that would substantially reduce the overhead of swapping tasks.

Another tasking improvement would be to allow the part of TCP that sends packets to run in more than one task. Telnet would call it as a procedure in the user task when the user types a character, but it would run in the send task when the retransmission timer went off. Only one task would have to run for many outgoing packets rather than two. These two improvements would eliminate most of the very small efficiency advantage that the procedure-based scheduling has over tasking.

From the user's standpoint, the ideas presented so far in this section would have only a minor effect on protocol performance. Are there any changes that we can make to substantially improve performance?

One idea is to replace the personal computer serial line interface, which takes one interrupt for each byte that traverses the serial line, with an Ethernet[8] or other high performance network interface that takes a single interrupt per packet. Table 6-4 suggests that character echoing time between two personal computers is mostly serial line time. A high performance interface could improve the character echo times between two personal computers by as much as a factor of two or three. The relative improvement would be somewhat less with character echoing from foreign hosts, since proportionately less of foreign host echoing overhead is serial line overhead. However, improvements with foreign hosts will be in the range in which the user most notices performance improvements.

A high performance network interface would not greatly improve the handling time for a screenful of data. Section 6.2.3 shows that replacing the serial line with a perfect network interface would decrease the handling time for a screenful of data from 2.8 seconds to 2.3 seconds. Making the protocols more efficient also seems to be a hard way to improve the handling time for a screenful of data since the protocol layers accounted for less than 0.2 seconds of the total 2.8 second handling time.

---

[8] Ethernet is a trademark of the Xerox Corporation.

Greatly improving handling time for a screenful of data probably is not possible. The I/O routines can only print data at a rate of 7400 bits per second. (Again, see Section 6.2.3.) The I/O routines seem to be reasonably efficient. The personal computer's processor speed appears to be the factor limiting the handling time for a screenful of data.

7.2 Topics for Further Research

The previous section suggests a number of areas for further research. Will the addition of an Ethernet card to the IBM Personal computer improve character echoing performance but not the handling time for a screenful of data? Is the speed with which a personal computer can perform terminal I/O an important criterion for choosing a personal computer for a remote login implementation? Our research suggests affirmative answers to both of these questions, but these answers should be confirmed.

We spent a lot of time in this thesis discussing the relative merits of tasking and procedure-based scheduling. Building a remote login implementation using a procedure-based scheduler would be the best way to find out all the details of how such a system works.

The most difficult question posed by this thesis is how to control congestion so that a high speed network does not swamp a machine connecting it to a low speed network. This question is a part of the generally unsolved problem of controlling congestion in and between computer networks.

7.3 Summary of Results

The performance measurements of the IBM Personal Computer Telnet implementation presented in Chapter Six show that a desktop personal computer can support an efficient internet remote login implementation with the same protocols used by large mainframes.

The speed with which programs run on small computers often determines the programs' utility. We therefore built the remote login protocols to run as quickly as possible. We

followed a number of strategies toward this end. The TCP implementation was tailored to the specific needs of the Telnet remote login protocol. This enabled us to simplify the complicated TCP protocol in a number of ways that allowed its implementation to run more quickly. The various protocol layers shared the overhead of asynchronous action. In this way, the layers were often able to share packets and send one packet instead of two. The layers shared data as much as possible to prevent the overhead of excess copying.

A factor particularly important to the speed with which the implementation ran was the mechanism used to pass control between the protocol layer modules. A multi-layer remote login protocol that handles asynchronous events in more than one layer needs multiple threads of control to efficiently and modularly process data. We proposed two methods of structuring control that allowed multiple threads. One of these methods,tasking, employed a combination of procedures and processes to structure control while the other method, procedure-based scheduling, used a top level scheduling module that employed procedure calls exclusively. Both methods could have formed the basis of an efficient implementation. The procedure-based scheduler would have run marginally faster. Tasking was the more flexible option, but the remote login protocol did not need the added flexibility. We chose tasking for the demonstration implementation to preserve consistency between the personal computer protocol and a number of other local implementations, and because its added flexibility might be handy for implementing other protocols on the personal computer in the future. The demonstration implementation showed that a tasking package could run on a personal computer.

Overall, we were pleased with the performance of the demonstration personal computer remote login protocol implementation. On a Telnet connection between two IBM personal computers connected by a wire, the round trip delay time for a single character was 0.072 seconds at a serial line speed of 19,200 bits per second while the handling time for a screenful of data could was under three seconds for an effective data transfer rate of over 5000 bps. The character echoing times to other local computers were as low as 0.128 seconds, while screenfuls of data could be transferred in as little as 2.91 seconds.

The speed of the serial line interface was the performance bottleneck for echoing characters. A more efficient network interface might improve the character echoing time by as much as a factor of two or three. The speed with which the processor could perform I/O limited performance when the protocols processed large amounts of data.

The personal computer protocol implementation used about 80KB of memory. The 194 KB memory size of the IBM Personal Computer was sufficiently large that it did not pose any constraints on our implementation.

References

[1] Saltzer, J., D. Clark, and D. Gifford.
Annual Report of the Computer Systems and Communications Group.
In *M.I.T. Laboratory for Computer Science Progress Report*. Volume 19. Laboratory for Computer Science, Massachusetts Institute of Technology, 545 Technology Square, Cambridge, MA 02139, 1983.

[2] Wright, K.
A File Transfer Program for a Personal Computer.
Technical Memorandum 217, Laboratory for Computer Science, Massachusetts Institute of Technology, 1982.

[3] Postel, J.
Internet Protocol.
In *Internet Protocol Transition Workbook*. Network Information Center, SRI International, Menlo Park, CA, 1982.
ARPANET RFC 791.

[4] Postel, J.
Transmission Control Protocol.
In *Internet Protocol Transition Workbook*. Network Information Center, 1982.
ARPANET RFC 793.

[5] Postel, J.
Telnet Protocol Specification.
In *Internet Protocol Transition Workbook*. Network Information Center, 1982.
ARPANET RFC 764.

[6] *Technical Reference.*
First edition. International Business Machines Corporation, 1981.
Part of the Personal Computer Hardware Reference Library

[7] Postel, J.
File Transfer Protocol.
In *Internet Protocol Transition Workbook*. Network Information Center, 1982.
ARPANET RFC 765.

[8] Postel. J.
Simple Mail Transfer Protocol.
In *Internet Protocol Transition Workbook.* Network Information Center, 1982.
ARPANET RFC 788.

[9] Postel. J.
User Datagram Protocol.
In *Internet Protocol Transition Workbook.* Network Information Center, 1982.
ARPANET RFC 768.

[10] Sollins. K.
Trivial File Transfer Protocol.
In *Internet Protocol Transition Workbook.* Network Information Center, 1982.
ARPANET RFC 783.

[11] Clark. D.
Window and Acknowledgment Strategy in TCP.
In *Internet Protocol Implementation Guide.* Network Information Center, 1982.
ARPANET RFC 783.

[12] *Intel iAPX 86, 88 User's Manual.*
Intel Corporation, Santa Clara, CA, 1981.
p. 2-51

[13] Cerf, V. and P. Kirstein.
Issues in Packet-Network Interconnection.
Proceedings of the IEEE 66(11). November, 1978.
p. 1400.

[14] Grange, J., and M. Gien, eds.
Flow Control in Computer Networks.
North Holland, 1979.