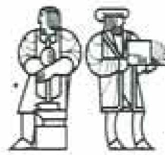
LABORATORY FOR COMPUTER SCIENCE



MASSACHUSETTS INSTITUTE OF TECHNOLOGY

MIT/LCS/TM-217

A FILE TRANSFER PROGRAM FOR A PERSONAL COMPUTER

Karl D. Wright

April 1982

545 TECHNOLOGY SQUARE, CAMBRIDGE, MASSACHUSETTS 02139

A File Transfer Program for a Personal Computer

by

Karl D. Wright

April, 1982

Copyright (C) Massachusetts Institute of Technology, 1982

This research was supported by IBM through discretionary funding made available to the M.I.T. Laboratory for Computer Science.

Massachusetts Institute of Technology

Laboratory for Computer Science

Cambridge Massachusetts 02139

Abstract

Thesis Title: A File Transfer Program for a Personal Computer

Author: Karl D. Wright

Thesis Adviser: Dr. Jerome Saltzer, Professor of Computer Science

This thesis explores engineering decisions involved in implementing a network file transfer program on a personal computer in response to criteria of low cost and reasonable efficiency. The issues include choice of hardware, design of the network, choice of implementation language, choice of communication protocols, and choice of software structure. A machine level protocol is designed. A project incorporating these and other ideas is undertaken and the ideas thus evaluated. Insight is gleaned into the performance expected under varying operating system and interrupt environments. A notion of an "ideal" operating system interface for applications similar to file transfer (which can exploit concurrency) is developed. Finally, possible improvements on the actual project are suggested based in part on the efficiency data obtained.

Keywords:

file transfer, personal computer, concurrency

Table of Contents

Introduction....................5

Chapter 1: Design....................7

A. Network and Low Level Protocol....................8

B. Transfer Protocol Choice....................14

C. Transfer Program Design and Structure....................16

Chapter 2: Performance....................30

A. Crude Timing....................30

B. Profiling....................34

Chapter 3: Conclusions....................37

A. Other Operating Systems....................37

B. Ideal Transfer Program Environment....................39

C. Possible Improvements....................40

D. Acknowledgements....................41

References....................42

Figures

1. Personal Computer Network Design.....9

2. TFTP Timing Diagram.....18

3. Code Structures.....21

4. TFTP Structure.....24

5. TFTP Structure with Packet Reassembly.....26

6. Graph of Time vs. Transmission Rate.....33

7. Profile Results of TFTP.....35

Introduction

Personal computers are in the process of revolutionizing many aspects of modern life. Information exchange is one area which will be affected greatly in coming years. As personal computers become commonplace, it is likely that people will wish to connect them together to form networks in order to send mail, transfer programs and data, and use community-owned mass storage devices or other such localized, expensive hardware. Some way of transfering files is a necessary means towards accomplishing this goal.

In this thesis, I will examine some of the engineering issues involved in implementing a file transfer protocol on a personal computer. A protocol is a concrete and thoroughly specified set of actions which allow different computers (or <u>hosts</u>) to communicate intelligibly via a network. Some of the engineering criteria which I attempted to conform with are:

1. File transfer must be straightforward, applicable to a large number of existing networks and machines, and reasonably efficient.
2. The network used must be low cost and hardware for it must be readily obtainable.
3. The protocol(s) used must be supportable by a wide range of hardware, e.g., the protocol(s) cannot require certain forms of concurrency since some personal computers do not have that capability.

Some of the more abstract issues that arise are the following:

1. What should be done to handle the case of a host which is not there some of the time?
2. How can the transfer program be structured to allow maximum concurrency between independent activities, and hence maximum efficiency?
3. What kind of efficiency can be expected using this structure under various kinds of operating system environments?

Finally, I will examine the insight this project gave into what personal computer hardware and operating systems should be like in order to support efficient network file transfer protocols.

Chapter 1: Design

In this chapter I will recount some of the design decisions made in order to implement the file transfer program. Decisions were made on the following questions:

1. What machine should be used?
2. How should a local network be constructed from these machines?
3. What protocol(s) should be used?
4. How should the transfer program be structured to allow maximum possible efficiency?
5. In what language should the transfer program be written?

All of these decisions had to be made in the light of the following considerations:

1. Overall cost must be low.
2. Hardware must be readily available.
3. Compatibility with large networks already built should be maintained.
4. The program should be readily transportable to other machines.
5. The hardware should be amenable to the project, i.e, the resulting program should have the possibility of being reasonably efficient and not be overly constrained by the hardware.

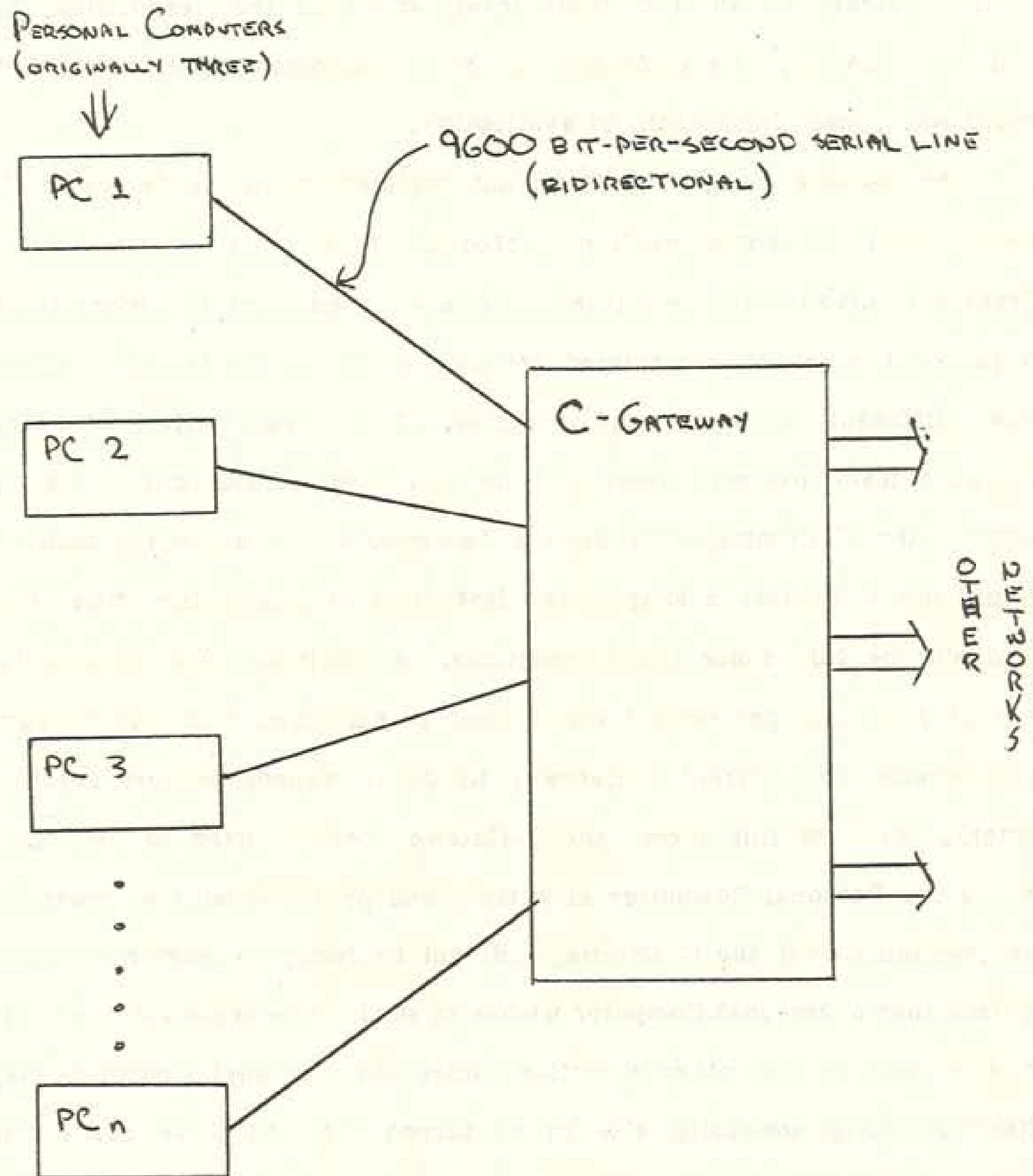
The machine chosen was the new IBM Personal Computer. The features that made it attractive for this project are centered around its ability to support concurrent peripheral communication activities. In particular, the disk controller is of the direct memory access type, which means that the processor is free to do other things while data is being read from or written to the disk. This is important because it allows the processor to concern itself with the network while disk access is in progress. In addition, this machine comes with a standard serial data port capable of running at speeds up to 56 thousand bits-per-second, which is quite fast enough to support a network.

This led immediately to the choice of the RS-232 serial line as the networking medium. Serial ports are standard on most personal computers and are thus easily obtainable at very low cost. This satisfies all the engineering criteria mentioned concerning networking medium.

A. Network and Low Level Protocol Design

The current network design is due in large part to Dr. Jerome Saltzer and Wayne Gramlich. It was decided to use a gateway already under development to do packet switching among Personal Computers or between a Personal Computer and outside networks. A gateway is a machine whose sole purpose in life is to route packets of data from network to network. Our gateway was dubbed the C-Gateway, since all the software for it is written in the C language.

(Figure 1: Design of Personal Computer Network)



Since this configuration requires one connection point for each Personal Computer attached to the network, the load on the C-Gateway increases quickly to an intolerable level, and thus the design does not scale up readily. Nevertheless, it is a reasonable interim solution, and it was chosen for reasons of availability.

The network design has important implications in the design of the lowest level of communication protocol. This level of protocol is directly involved with the actual hardware, in contrast to higher levels of protocol which are considered standard and thus are the same across many different hardware configurations. The design criteria for this hardware-level protocol rests on the following considerations: First, because the C-Gateway will have a tendency to be extremely busy, it should not have data coming in too fast or at an inopportune time since data will be lost under these conditions. A maximum data transmission rate of 9600 bits per second was chosen to help meet this criteria (and also because the current C-Gateway hardware cannot support anything faster). Even at this speed, the C-Gateway cannot listen to more than one or two Personal Computers at a time, and provision must be made for the possibility that the C-Gateway will not be ready to receive a packet of data that a Personal Computer wishes to send. The argument can apply equally well in the other direction, since the Personal Computer may either be doing something else or be turned off. In these cases, the packet is undeliverable and the C-Gateway should be aware of it, in case error actions need be undertaken. In addition, the protocol must be prepared to handle lost bytes, since the communication media can occa-

sionally lose characters due to noise or synchronization problems. Finally, the protocol should support simultaneous data transmission in both directions for reasons of efficiency and generality. The following ready-acknowledge protocol fulfills these criteria:

Machine A wants to send a packet of data to machine B (A and B are directly connected to each other by an RS-232 serial line):

1. Machine A sends B a RDY request. "Are you listening?"
2. Machine B sends A an ACK reply. "I'm ready; let 'er rip!"
3. Machine A sends B a data packet encoded in such a way that RDY, ACK, or END never appear in the data. "Here it is."
4. Machine A sends B an END signal. "I'm done."

The encoding procedure used on the data is known as <u>character stuffing</u>. One character is designated as a prefix character, and all occurrences of that character or any of the signals are translated into the prefix character followed by another (non-signal) character.

If machine A sends a RDY but does not receive an ACK for too long a period of time, it retransmits the RDY and repeats once or twice, in case either the RDY request or the ACK reply was lost. If it is still unsuccessful, it assumes machine B is dead or not interested in talking.

If machine B sees a RDY in the middle of data, it assumes that the entire packet was sent, but the END signal was lost. If machine B is receiving data, and the data stops coming for long enough, machine B

assumes that the END signal was lost but the packet is complete. [Any errors in this assumption will be detected by a higher level of protocol.] If machine B sees an END signal while it is expecting data, it knows the packet is complete. If machine B sees an ACK reply in the middle of data, it sets a flag so that the transmitting side knows that an ACK reply has been received. This is how bidirectional simultaneous transfers are supported. A typical bidirectional transfer looks (or sounds) like this:

1. Machine A is sending data to B.
2. Machine B decides it wants to send data to A at the same time, so it sends A a RDY request.
3. Machine A sees the RDY request and immediately stuffs an ACK reply in with the data it is sending.
4. Machine B sees the ACK among the data it is receiving and begins sending its data to A.
5. Both transfers continue simultaneously.

Notice that either side can enforce monodirectional communication by simply not responding to RDY requests until no packet is in the process of transmission.

The interface to the Low Level Protocol (LLP) was designed with a wide range of applications in mind. For example, applications requiring parallelism mean the interface has to to have the characteristic that a program trying to send or receive data never gets "stuck" waiting for a

packet to be sent or received, or waiting for the other machine to acknowledge a request. This is necessary to take full advantage of the bidirectionality of the protocol, since a program may want to send and receive packets simultaneously and therefore must pay attention to both the sending line and the receiving line. In the case of a file transfer protocol, the program may wish to do disk access and receive or send a packet simultaneously. The interface chosen for the task is the following:

receive_packet_status_check:	Returns a flag indicating whether or not a packet has been completely received and is waiting. Side effect: Performs functions of LLP.
get_received_packet:	Waits for a packet if there is not one already there, and returns the packet received.
send_packet_status_check:	Returns a flag indicating whether or not a packet has been completely sent and routines are ready to handle another. Side effect: Performs functions of LLP.

send_packet: Waits until the last packet is completely sent, then starts sending a new one.

This interface satisfies the criteria mentioned above. In addition, it is important to note that the function of the LLP is not performed by the routines which actually receive and send individual characters on interrupts. The decision not to do LLP work on individual character send or receive interrupts came about for two reasons. First, part of the function of the LLP is to allow programs who do not wish to receive data to simply ignore the existence of the network. Second, implementing LLP at interrupt level would require extensive and complex assembly code, which is difficult to maintain and could eventually interfere with the ability to run the communication lines at a faster rate, since the maximum data transfer rate is strongly dependent on the the delay inherent in the interrupt handler. Nonetheless, as I will mention later on, it may eventually prove worthwhile to implement LLP on the interrupt level. In this case the interface need not be changed, but the status routines will no longer have to be called to carry out the function of the LLP.

B. Transfer Protocol Choice and Implementation Considerations

The file transfer protocol chosen for this project is the Trivial

File Transfer Protocol (or TFTP) [1]. This protocol is considered "trivial" because of its simple flow control and total ignorance of complex issues like protection and access. It is a lock-step protocol, which means that only one packet of data is sent at a time, and an acknowledgement for that packet must be received before the next packet is sent. Thus TFTP is inherently a monodirectional protocol, i.e., only one packet is in transit at any given time. This protocol was chosen because of its relative ease of implementation, its widespread use among the networks at MIT, and its trivial treatment of protection issues.

TFTP is built on top of the User Datagram Protocol (UDP) [2], which, in turn, is built on top of the Internet Protocol (IP) [3], which also happens to be the Defense Department standard. These protocols provide packet checksums and a length (which is redundant in view of the low level protocol) for reliable communication. Thus, TFTP has high useability and is fairly straightforward to implement.

Two real choices were possible for the selection of the programming language in which to write TFTP for the Personal Computer. The language PASCAL is distributed by MicroSoft for use on the Personal Computer, but, unfortunately, it turned out to be inefficient as a compiler and did not have the ability to interface to assembly language at the time the project was undertaken. This was a serious problem since there was apparently no way to access the serial port from PASCAL. The other choice was the C language. Traditionally, C compilers produce highly efficient code which has ready access to assembly language. The major drawback to the use of C as the medium in which to construct TFTP was

that all the software development tools would have to be written from scratch, although most of the compiler itself could be generated by modifying code already written. Nevertheless, this was the course chosen, and thanks to the effort of Chris Terman, Wayne Gramlich, John Romkey, and David Bridgham, a full set of software tools for the C language on the Personal Computer were available in time to complete this paper.

C. TFTP Design Criteria and Resulting Program Structure

The principal problem with implementing TFTP on the Personal Computer is one of efficiency. Both the serial communication line and the disk are rather slow relative to the speed of the processor. Hence, any sort of parallelism that exists must be taken advantage of if a reasonable speed is to be attained by the end program. In particular, disk access and packet input or output must overlap in time as much as possible to squeeze out the maximum data transfer rate. However, as mentioned before, TFTP is a lock-step protocol, which means that only one packet can be processed at a time. That somewhat limits the maximum possible overlap achievable. The diagram on the following page illustrates various timing possibilities. It illustrates three levels of concurrency: none, total, and partial. Partial concurrency (as drawn) is based on two assumptions:

1. No network access can start while a disk access is in progress.
2. Packet reception is not under interrupt control, so reception cannot occur during disk access.

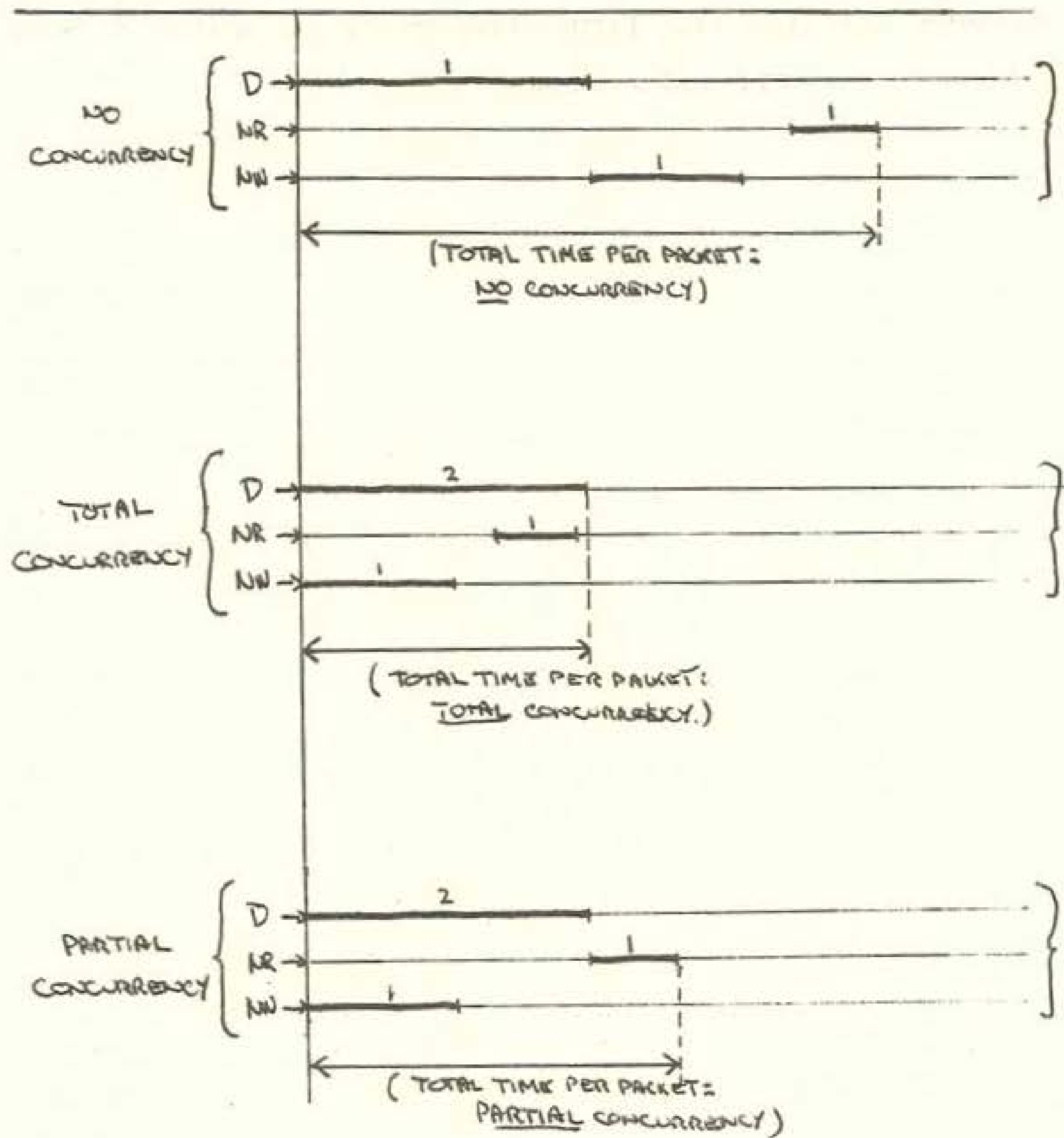
These happen to be the conditions which prevail in the Personal Computer and LLP, thus partial concurrency (as drawn) is what is achieved in this TFTP implementation.

(Figure 2: Timing diagram)

MOVING DATA FROM DISK TO NETWORK:

[D ≡ DISK READS, NR ≡ ACKNOWLEDGE PACKETS, NW ≡ DATA PACKETS]

NUMBERS INDICATE TFTP PACKET NUMBERS.



MOVING DATA FROM NETWORK TO DISK IS SIMILAR: ALL PACKET WRITES BECOME ACKNOWLEDGE PACKETS, ETC.

Another design consideration for TFTP is the relative speed of the TFTP software itself. Long delays in packet encoding and decoding will result in the loss of overlap and hence a reduction in the rate of data transfer. One heuristic measure often used for gauging efficiency of network software is the total number of times the data need be copied between the time it resides on the disk and the time it is actually moving around on the network. A lower number of copies signifies a program which is likely to behave more efficiently than one with a higher number of copies. Thus the software structure should be chosen so as to reduce the number of copies wherever possible.

The pertinent design criteria are summarized below:

1. The design should permit a maximum amount of control over concurrency between independent processes.
2. It should encourage as few copies as possible.
3. It should be flexible enough to allow for later changes in system configuration, e.g., an increase in speed of the serial line or disk, or a restructuring of the operating system.

There are two logical choices for overall system structure:

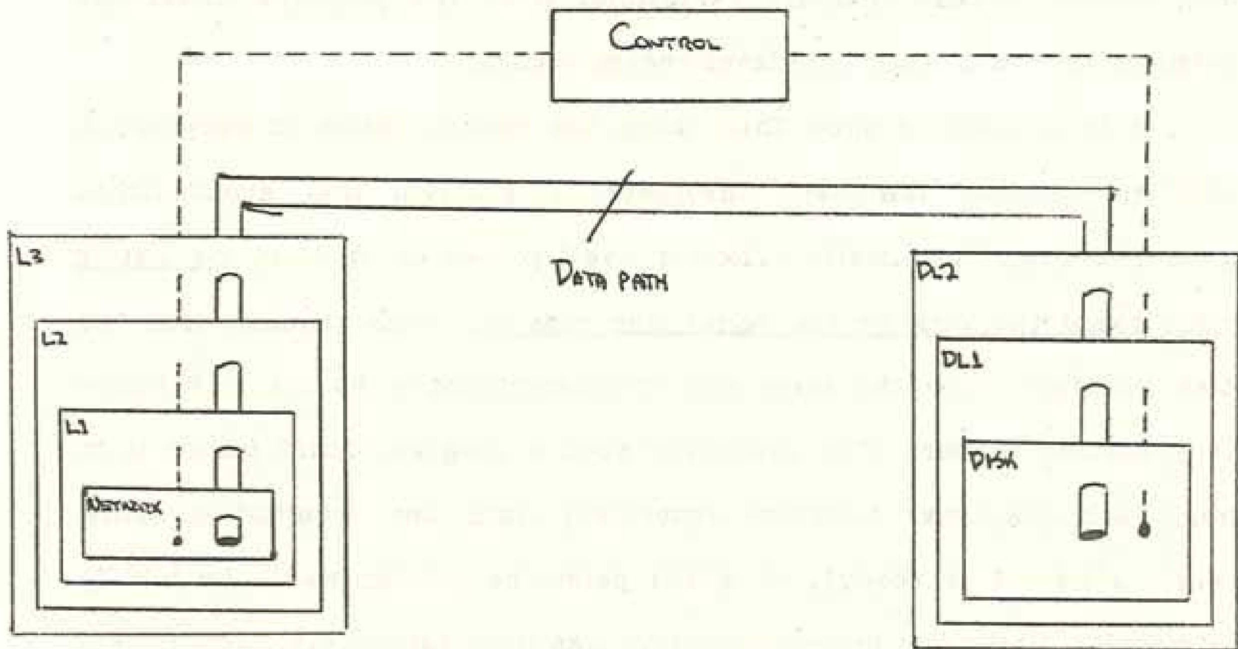
1. Packet handling is performed by each distinct protocol layer individually. As an example, suppose the TFTP layer wants to send a packet of data. It contructs the packet, then hands it to the UDP layer with the instructions, "Here's a packet. Send

it to such and such a network address, and tell them so and so sent it." The UDP layer takes the packet, adds some information to it, then turns around and wakes up the IP layer, who, in turn, adds still more information, and then proceeds to hand the packet to the LLP for transmission.

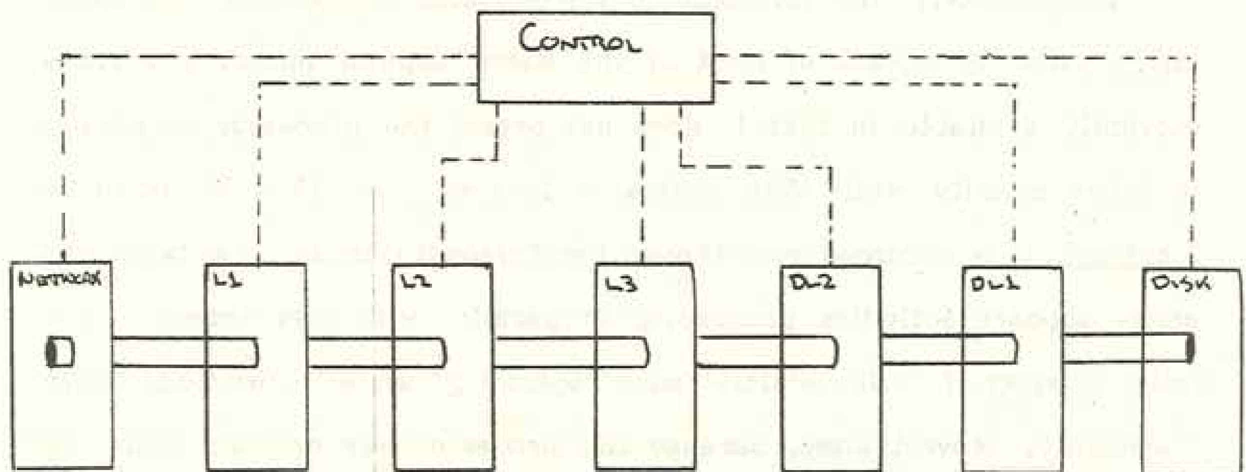
2. Packet handling is done by a sort of "cradle-to-grave" packet tracking system, similar to a finite state machine and data path. The top layer explicitly performs all operations on the packet, and all the lower layers simply perform alterations <u>without moving the packet around</u>. As an example, suppose the TFTP control layer wishes to move a packet of data from the disk to the network. It first calls the disk driver with the instructions, "Get me such an such a block of data." The disk driver fetches the data, at which point the TFTP control layer tells another routine to turn it into a bona-fide TFTP-UDP-IP packet (note that this procedure in itself may involve nested calls to several layers). Finally it tells the LLP to transmit it. Note that the data flow is explicitly controlled by the topmost layer (or <u>control</u> layer) in this kind of structure, and note furthermore that modularity and layering need not be violated by using this approach.

(Figure 3: Code structures)

(1) CONVENTIONAL "LAYERING" OF PROTOCOLS:



(2) "CRADLE TO GRAVE" PACKET SYSTEM:



Of the two choices, only the second fulfills the design criteria. The first not only promotes more copies, but makes the overlap of disk and network access virtually impossible, since the principle difference between the two is where low level access occurs.

It is possible to show that, using the second option in conjunction with the correct low-level interfaces, a program that accomplishes generalized and maximally efficient overlap can be written, no matter what speed the disk or the serial line runs at. This requires that the disk interface have the same sort of characteristics as the LLP interface defined earlier. The procedure such a program would follow is to poll each low-level interface repeatedly until one required servicing (e.g., a packet is ready), at which point the polling would be briefly interrupted while the necessary action was undertaken (e.g., the packet is decoded and disk writing is started).

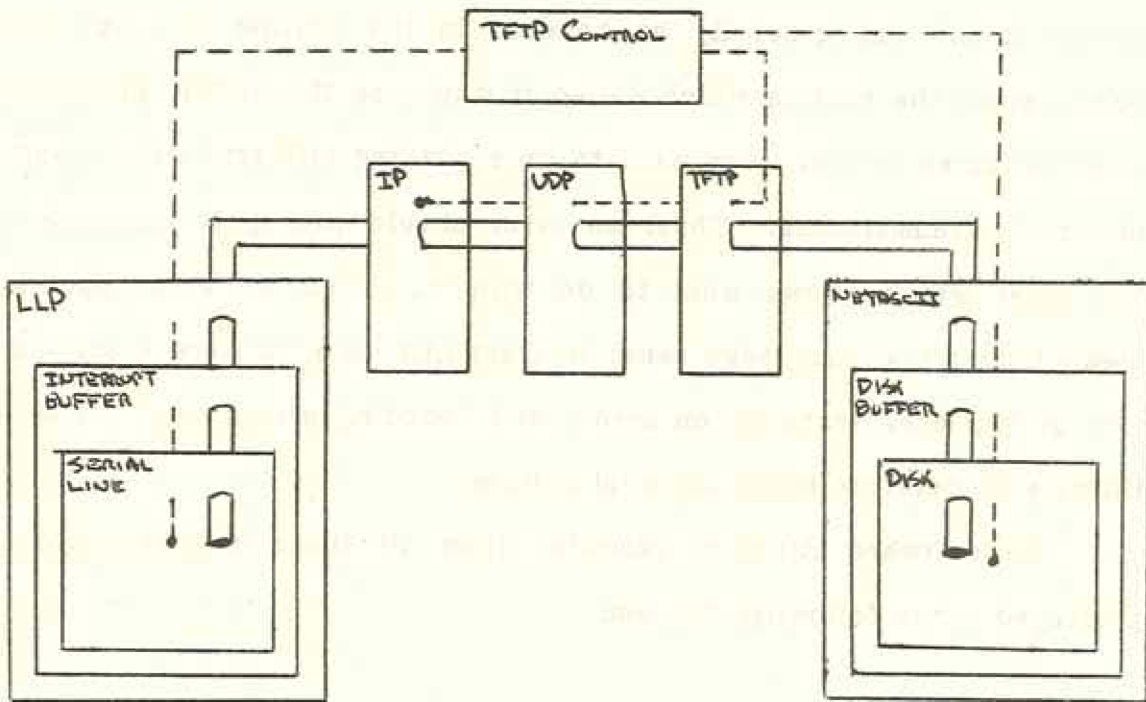
Unfortunately, the interface to the Personal Computer's disk operating system is typical of most of the microcomputer operating systems currently available in that it does not permit the processor to engage in other activity while disk access is in progress. This is known as blocking. It is enforced even though the Personal Computer hardware can easily support activities proceeding in parallel with disk access. In a later chapter I will explore such operating system questions more thoroughly. Nevertheless, because the processor can be interrupted by the serial line while it is waiting inside the operating system for a disk operation to be completed, a high degree of overlap can still be attained, since (as is apparent from the timing diagram) most potential

overlap for TFTP occurs between individual packet transmission and disk access.

One can make a structure argument about error handling as well. Errors should not generally be handled in the routine in which they occur, since the routine which called it may need the knowledge that an error occurred or may even want to do something different with it under different circumstances. Thus, an error should propagate outwards to the layer which knows what to do with it. From an error handling viewpoint, it does not make sense to treat LLP as if it were a subfunction of IP, since transmission errors and decoding errors mean different things and therefore imply different actions.

The software structure resulting from all these considerations is portrayed in the following diagram:

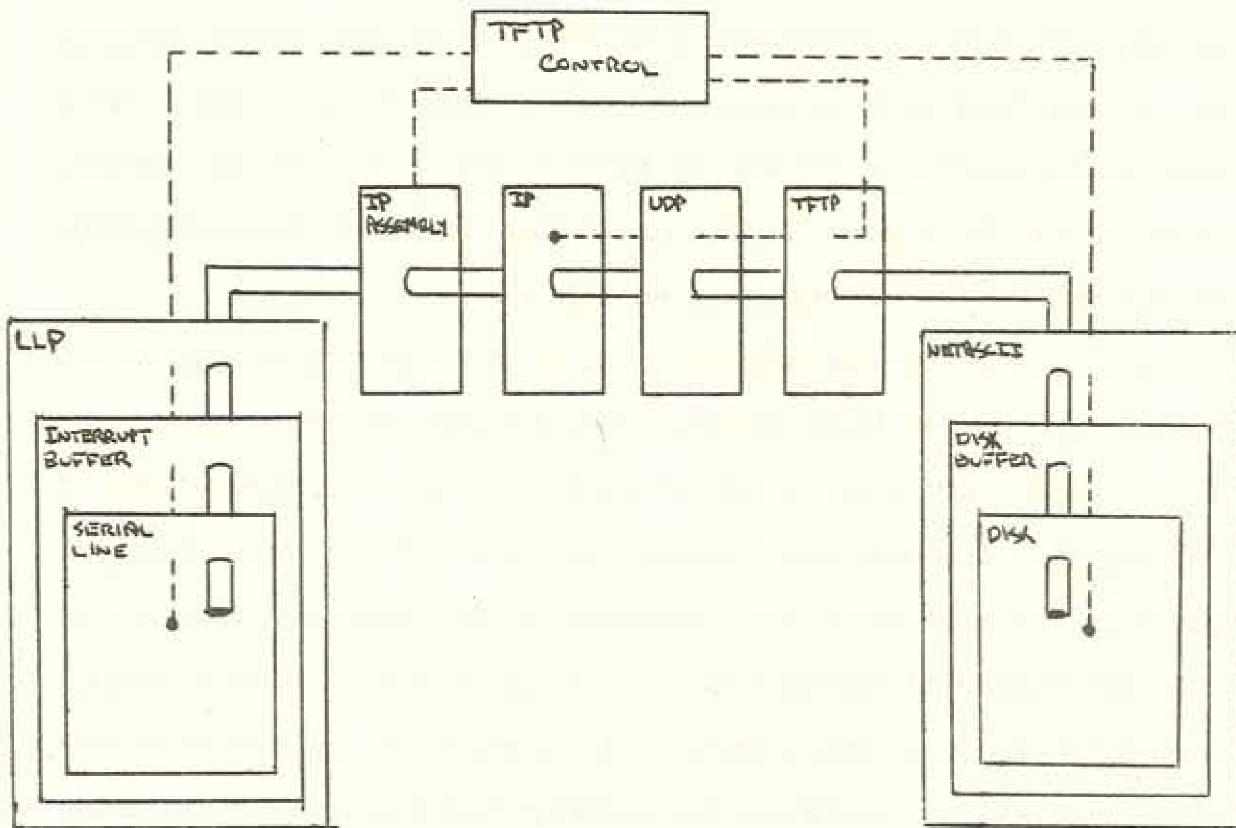
(Figure 4: TFTP software structure)



Notice that the total number of copies required for the implementation is minimized: One copy is performed by the operating system, one copy is performed in order to encode and decode netascii, and a third copy is performed to encode or decode LLP packets. This copy could later be eliminated by performing LLP function at interrupt level, as mentioned earlier. The fourth and final copy involves moving the packet from the memory to the serial line.

An issue which was ignored in the actual implementation is packet reconstruction after fragmentation. The Internet Protocol, upon which TFTP is eventually constructed, allows for the possible fragmentation of packets when they are routed through networks with a constrictive upper bound on maximum packet size. Although all the networks in local use at MIT are capable of handling the largest possible TFTP packet without fragmentation, it is conceivable that sometime in the future it will become necessary to implement the packet reconstruction feature. Packet reconstruction implies that another copy need be performed in order to receive a complete packet. This, in turn, implies the following additions to the TFTP control path and IP layer:

(Figure 5: TFTP software structure with packet reassembly)



The packet reassembly module should be treated (from the outside) as a device which is always ready to accept packets but which may not produce a packet as a result. This differs slightly from the interface specified for the IP, UDP, and TFTP modules in that data is actually copied, and thus, the input packet is not guaranteed to be in the same place as the output one. This is an important point in light of the UDP checksumming algorithm, which accesses data in the Internet header [2]. Therefore packet reassembly cannot be accomplished cleanly alongside normal internet packet decoding. The best interface for the reassembly module looks like this:

assemble_packet: Takes a packet, performs packet reconstruction algorithm using hidden buffers and bitmaps, and returns either (1) A completely reconstructed IP packet (i.e., a pointer to an internal buffer), or (2) some signal indicated no packet is ready for output.

This interface hides a multitude of complexity from the TFTP controller, yet is also straightforward to implement. It can easily be integrated with TFTP by replacing all "get_packet" calls to the LLP with that same call followed by a call to the packet assembly routine. A signal indicating no packet is ready yet should be treated in the exact same way as is an illegal packet signal from the UDP or IP layers.

Another feature which has been written but not debugged or included

with the current TFTP is an implementation of the Name Server Protocol [4], intended as an enhancement to the user interface. The Name Server Protocol allows the TFTP user to reference remote machines by name rather than internet address. This feature should be easy to incorporate into TFTP once the C-Gateway is working, since it is completely self-contained and is structured in a way similar to TFTP.

This design discussion has completely ignored an aspect of disk systems known as synchronization. Because disks are not true random-access devices but instead rely on a form of sequential access (the ordering of sectors on a track), it is not necessarily true that disk accesses take the same time to complete in different situations. For example, since the operating system has been optimized to read files sequentially with little delay between read requests, it is entirely possible that far less time will be needed to read a file quickly into memory than to read blocks individually from disk. It is therefore conceivable that such gains could surpass any gains made due to concurrency, in which case it would be more efficient to first read a given file into main memory before transmitting it, even though there would no longer be any concurrency between these operations. This point was ignored (temporarily) for two reasons:

1. Any effects of this type are very strongly dependent on the actual kind of machine, e.g., a different kind of personal computer would have a different disk format where synchronization is not a problem.

2. A given user program using TFTP may not have room in memory to buffer the entire file, at which point the disk-to-memory transfer is done piecemeal. Because of the extra copy required under this scheme and the lack of any sort of concurrency, it is unclear at which point one approach (concurrency) becomes more efficient than the other approach (copying to main memory first). This is an area where further research need be done.

Chapter 2: Performance Considerations

In this chapter I will document the performance tests done on the TFTP implemented as described in the previous chapter. Each performance test description will be accompanied with expected results and actual results obtained.

All performance tests were carried out between two Personal Computers for two reasons. First, connection to the actual network can introduce delays which are not dependent on the Personal Computer TFTP implementation, and hence would have to be factored out of any performance evaluation. Second, the C-Gateway suffered a substantial implementation delay and was not ready for use at the time of this writing.

A. Crude Timing

This performance test is a simple timing of a typical file transfer. A 20-thousand byte file was transfered between the two Personal Computers and the total time required was logged. In addition, a number of minor modifications were made to the program on an experimental basis. First, the size of the interrupt-level transmit buffer was varied in order to gauge the amount of overlap between disk access and packet transmission. The expected correlation is that as the interrupt character transmit buffer gets smaller overlap decreases, and thus the

amount of time needed to complete the transfer increases, approaching some maximum value (which represents no overlap). By looking at the maximum and minimum times, it is possible to judge just how much overlap is occurring.

Forty-five seconds were required to transfer a 20-thousand byte file in the case where the interrupt transmit buffer was large enough to contain an entire TFTP packet and the speed of the line was 9600 bits-per-second. This translates into an effective data transmission rate of 4400 bits-per-second on the serial line, out of a theoretical maximum of 8500 bits-per-second. The theoretical maximum is based on the size of the TFTP-UDP-IP packets in relation to the amount of file data actually contained within. In particular, to move a given block of 512 bytes of actual data, at least 576 bytes of data must be transferred between the two communicating machines (544 in the data packet and 32 in the acknowledge packet). Thus, about 10% of the total 9600 bit-per-second bandwidth of the serial line is used up by protocol overhead. Of the remaining 90%, roughly 40% is wasted because of the combination of software delay and disk delay.

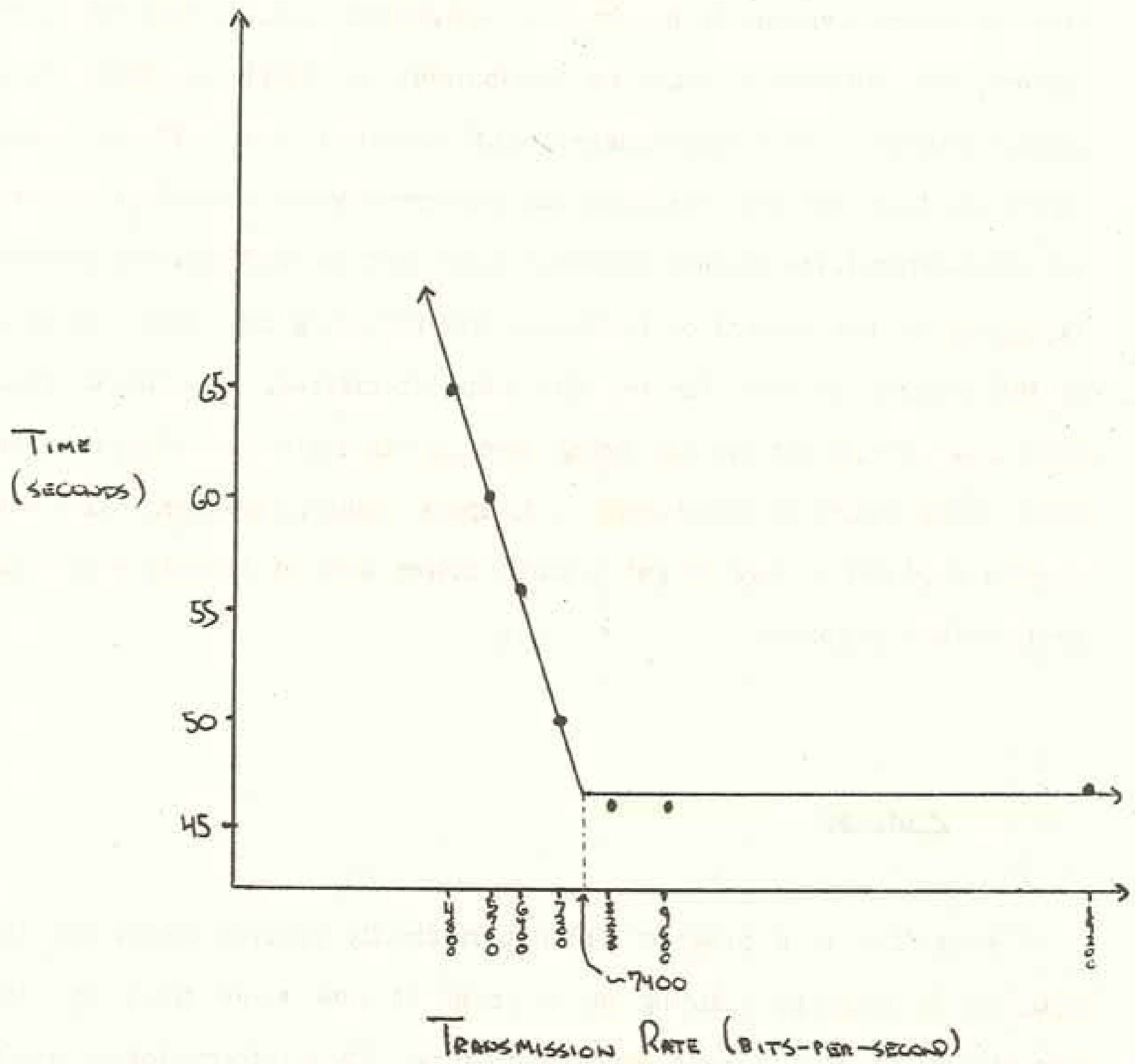
Notice that TFTP achieved only 3500 bits per second as far as the actual data was concerned, since the serial line uses ten bits to transmit each 8-bit byte. This is a point of clarification only, since my entire analysis will treat bytes as having ten bits each and thus ignore the effects of the underlying serial line protocol.

With a minimal interrupt transmit buffer, TFTP took substantially longer to transfer the same file. Under these conditions, the total

time required turned out to be about 65 seconds. Thus, 20 seconds are required for actual packet transmission and the rest of the time is used in software and disk delays. These numbers indicate that, assuming there are no software delays, the data transmission rate can be cut almost in half without affecting the speed of the TFTP transfer in the least, since it is apparent that (at 9600 bits per second) data transmission takes about half the time of disk access.

The next step was to verify that the serial line speed could be reduced without affecting TFTP in any way. The interrupt transmit buffer was restored to full size and data speeds other than the 9600 bits per second already documented were tried. Two that were originally chosen are 19,200 bits per second and 4800 bits per second, doubling and halving the original data rate. The 19,200 rate turned out to be equivalent from the performance viewpoint to the original 9600 rate, as one would expect from data already gleaned, but the 4800 rate caused TFTP to run substantially slower. In order to get a handle on exactly how much delay is due to disk access and how much is due to software, I decided to choose other numbers between 4800 and 9600, in particular, 8228, 7200, 6400, and 5760 bits-per-second. The transfer rates obtained are portrayed on the following graph. Note that there is a sharp cutoff where transmission rate begins to matter at roughly 7000 bits-per-second.

(Figure 6: Performance vs. Transmission Rate)



Notice that it is now possible to estimate the portion of time which is due to TFTP-UDP-IP software overhead. Since the actual transmission rate is known to be about 4400 bits-per-second, and the data rate at which overlap is maximal is somewhere around 7000 bits-per-second, the difference must be attributable to TFTP overhead (i.e., packet headers and acknowledges), LLP overhead, and software delay. TFTP overhead accounts for about 600 bits-per-second, while LLP accounts for an indeterminate amount anywhere from zero to 5000 bits-per-second, depending on the amount of character stuffing going on. This number is in the vicinity of zero for the file being transfered. Therefore, about 2000 bits-per-second are not being used due to software delay, or about 20%. This figure is necessarily a ballpark figure, however. The next section explores a way to get a much better idea of exactly where the performance is going.

B. Profiling

A profiler is a program which periodically records where the test program is running, building up a count of how many times the test program is caught in each area of memory. This information is useful because it describes exactly how much time is used performing a given task within the program, with few exceptions. The profiler used on this implementation of TFTP is based on a clock which interrupts execution 18.2 times per second. This rate is fortunately not a multiple of the

baud rate clock (which drives character transmission and reception) or the disk drive spin rate. Thus it must produce an accurate picture of the operation of TFTP, since it is asynchronous to activities performed within TFTP.

Nevertheless, because the profiler is interrupt driven, it cannot report the amount of time spent in the interrupt routine or any other routine which disables interrupts. Instead, time used for interrupts is logged in the routine which was executing at the time of the interrupt.

The profile results of the typical transfer as mentioned above are as follows:

(Figure 7: Profile Results)

Function	Time spent	Percentage of total
Operating system	364 units	43 %
Low level protocol	234 units	28 %
File I/O library	122 units	14 %
Clock routines	34 units	4 %
TFTP control layer	33 units	4 %
Packet encoding	32 units	4 %
Netascii encoding	29 units	3 %
Total:	848 units	= 46 seconds

Notice that the total file system delay accounts for 57% of the processor time, whereas actual TFTP work accounts for only 11%. I separated out the clock routines for the reason that they are used mainly during packet transmission and reception loops, and thus deserve to be lumped in with the LLP time.

Profiles were also done of the same standard transfer with the serial line running at 19,200 bits-per-second and 4800 bits-per-second. The former differed in no appreciable aspect from the numbers in the table above, while the latter showed a more even distribution between time spent in the operating system and time spent waiting for packets in the Low Level Protocol.

Notice that these results do not give any conclusive indications of the efficiency of LLP except for an upper bound on the amount of time spent doing LLP work. This is because time spent waiting for packets to arrive or characters to be transmitted also is a part of the LLP profile count. Nevertheless, less than 12 seconds are actually spent doing LLP work (this is 28% of 46 seconds). Since 22,500 characters are being transmitted, less than 530 microseconds are required per character, which translates into a data transmission rate of 19,000 bits-per-second. This number represents a lower bound on the maximum data transmission rate which LLP can keep up with. Thus, LLP is a good protocol to use up to and somewhat beyond a transmission rate of 19,200 bits-per-second.

Chapter 3: Conclusions

In this chapter I will summarize what I have learned from the TFTP project that relates to issues raised in the introduction of this thesis. In particular, I will cover the following:

1. What can be expected in different operating system environments with or without interrupts.
2. What the ideal operating system environment is for a maximally concurrent TFTP implementation.
3. What improvements can be made to the file transfer program as it now stands.

A. Other Operating Systems

As was apparent in the previous two chapters, the only concurrent activity that Personal Computer's operating system allows is interrupts. This resulted in partial overlap between disk access and packet transmission and caused a 25% reduction in data transmission time, for the case where the Personal Computer was buffering an entire packet and running the serial line at 9600 bits-per-second. The result was that a transfer of 20,000 bytes took 45 seconds, or roughly 2 milliseconds per data character.

It is easy to imagine an operating system in which _no_ concurrent activity is allowed. Systems like this are widespread; examples are the TRS-80 and the APPLE, to name but a few. In these systems, no overlap is possible, and the best one can hope for is about 65 seconds to transfer 20,000 bytes of data. This number was obtained by eliminating all concurrency from the Personal Computer implementation (as described in Chapter 2).

It is also possible to imagine a system which has very little operating system delay. With the advent of Winchester technology, medium-speed hard disks are becoming cheaper and more readily available. In a system like this, the major bottlenecks are the serial line and the software, in that order. The software delay costs about 11% of the total time on the Personal Computer. This delay, which comes to about 5 seconds, is dependent only on the quantity of data being transmitted. Therefore the time required would be 22.5 seconds (serial line delay for 20,000 x 576/512 characters) + 5 seconds (software delay), or about 28 seconds for 20,000 bytes of data.

A system with much faster or slower communications is also a possibility. The timing tests show conclusively that data rates faster than 7000 bits-per-second do not significantly improve performance, given the operating system constraints of the Personal Computer. Markedly slower rates have not been an emphasis of this thesis, but one would expect performance to drop off linearly with decreasing data transmission rate, since the disk and software delays are essentially fixed.

B. The Ideal Environment

The ideal operating system environment for TFTP must promote maximum parallelism. To do this, an operating system with no explicit multiple processes must support independent peripheral activities without blocking. Thus the format of all peripheral interfaces should have two parts:

1. A status routine: "Are you ready to perform the function?" or "Are you done performing the function?"
2. An execution routine: "Perform the function." This routine performs blocking if the device is not ready.

For example, the disk read interface might have a routine "Read" and a routine "Read_done?" which initiate a read and notify of completion, respectively.

Preferably, the status routine should have nothing to do with actually performing the function, since it is confusing to have to specify that such-and-such a status routine must be called every so-and-so seconds or the world blows up, or, more probably, the function does not get performed. This premise is violated in the current TFTP implementation by the LLP; the status call does indeed perform LLP function. The reasons behind the decision to do it this way are documented in

C. Possible Improvements

The following, in order of importance, is a list of possible improvements which could be made to TFTP, LLP, UDP, IP, the interrupt environment, or the operating system enviroment that would enhance either efficiency or useability:

1. Replace the current status-call-driven LLP with an interrupt-driven version of the same protocol. This buys useability in the low-level communication area, i.e., makes it easier to write other protocols or programs which use LLP. It also buys efficiency in two ways: First, packet reception can proceed concurrently with disk access (as already occurs with packet transmission). Currently, if the Personal Computer is busy with disk access and the C-Gateway wants to send a packet to it, the RDY request does not get acknowledged until the end of the disk operation. If LLP handling occurred at interrupt level, the RDY request would be acknowledged immediately and packet transmission could start. This can save time, especially on transfers from the network to the disk (see Figure 2). Second, one less copy need be performed, since character stuffing can be done on the fly by the interrupt handler.

2. Improve user interface of TFTP by completing the implementation of the Name Server Protocol package. This adds useability for the end user of TFTP.
3. Upgrade to an operating system and input/output package that has an interface of the type discussed in the previous section. This would buy efficiency.
4. Add packet reconstruction as detailed in Chapter 1. This buys useability by increasing the number of machines which can be reached through this TFTP implementation.

D. Acknowledgements

This concludes my analysis of the Personal Computer File Transfer Program project. I would like to thank Wayne Gramlich, Chris Terman, David Bridgham, and John Romkey for providing the necessary software tools, Larry Allen, Bob Baldwin, and Mike Patton for providing implementation hints and thorough knowledge of the subject, and, of course, Jerry Saltzer, who came up with the idea in the first place.

This research was supported by IBM through discretionary funding made available to the M.I.T. Laboratory for Computer Sciences.

References

[1] Sollins, K., "The TFTP Protocol," Massachusetts Institute of Technology, IEN 133, January 1980.

[2] Postel, J., "User Datagram Protocol," RFC 768, USC/Information Sciences Institute, August 1980.

[3] Postel, J., "Internet Protocol," RFC 760, USC/Information Sciences Institute, January 1980.

[4] Postel, J., "Internet Name Server," USC/Information Sciences Institute, IEN 116, August 1979.