The Desktop Computer as a Network Participant

JEROME H. SALTZER, FELLOW, IEEE, DAVID D. CLARK, MEMBER, IEEE, JOHN L. ROMKEY, AND WAYNE C. GRAMLICH

Abstract—A desktop personal computer can be greatly extended in usefulness by attaching it to a local area network and implementing a full set of network protocols, just as one might provide for a mainframe computer. Such protocols are a set of tools that allow the desktop computer not just to access data elsewhere, but to participate in the computing milieu much more intensely. There are two challenges to this proposal. First, a personal computer may often be disconnected from the network, so it cannot track the network state and it must be able to discover and resynchronize with that state very quickly. Second, full protocol implementations have often been large and slow, two attributes that could be fatal in a small computer. This paper reports a network implementation for the IBM Personal Computer that uses several performance-oriented design techniques with wide applicability: an upcall/downcall organization that simplifies structure; implementation layers that do not always coincide with protocol specification layers; copy minimization; and tailoring of protocol implementations with knowledge of the application that will use them. The size and scale of the resulting package of programs, now in use in our laboratory for two years, is quite reasonable for a desktop computer and the techniques developed are applicable to a wider range of network protocol designs.

Overview

THIS paper describes the issues encountered and lessons learned in the design, implementation, and deployment of a full-scale network protocol implementation for a desktop personal computer. The protocol family implemented was the United States Department of Defense standard Transmission Control Protocol and Internet Protocol [1], [2]. The desktop computer was the IBM Personal Computer attached to one of several local area networks: Ethernet, PRONET, and an RS-232 asynchronous serial line network. The collection of programs is known as PCIP.

The project was undertaken in December 1981, shortly after the IBM PC became available. Initial implementations using an RS-232 asynchronous serial line network were in operation in the Summer of 1982, and a complete implementation for the Ethernet was placed in service at M.I.T. in January 1983. Since that time the implementation has been polished, drivers for other networks have been added, the software has been used in many applications unrelated to network research, and the programs have been placed in service at several other sites. Enough experience with the implementation has been gained to provide a convincing demonstration that the techniques used were successful.

Introduction

A comparison of large mainframe computers attached to networks and similar machines with no networking capabilities makes a clear case for the value of network attachments. The value includes these abilities: to move files from one machine's file system to another that has better long-term reliability, more space, or cheaper storage; to use a unique printer that has better fonts or higher speed on another computer; to log in as a user on another machine to get to a different database manager or different programming language; and to send and receive electronic mail within a large community. Such abilities have all proven to be important extensions of the basic stand alone capability of a computer system. The desktop personal computer, whose main advantage lies in its administrative autonomy, potentially can be extended in value by network attachment even more than the large shared mainframe. The reason is that by itself it is likely to have a smaller range of facilities than does a large shared mainframe, and thus a mechanism that offers the ability to make occasional use of unique services found elsewhere is especially useful.

In the mainframe world, this added function has come with a substantial cost, however. Implementations of network protocols have usually turned out to be large and slow. Although size of software packages is of somewhat less concern than it once was (because the cost of memory to hold those large packages seems to drop faster than the packages grow) long path lengths through those packages can produce bottlenecks and limit data rates. For example, although the hardware links in the Arpanet are mostly of 56 kbits/s, few attached systems can sustain a data rate much above 15 kbits/s. When those same systems are attached to local area networks that can accept data rates of 10 Mbits/s their software continues to be a bottleneck in the 15 kbit/s area.

Manuscript received September 14, 1984; revised January 18, 1985. This work was supported in part by IBM Corporation under a general grant for computer science research to M.I.T. Many of the ideas were borrowed and some of the code was ported from projects supported by the Defense Advanced Research Projects Agency.

The authors are with the Department of Electrical Engineering and Computer Science, Massachusetts Institute of Technology, Cambridge, MA 02139.

0733-8716/85/0500-0468$01.00 ©1985 IEEE

Thus the question arises: can one put together a useful implementation of a network protocol family, one that fits into a desktop computer that does not have a virtual memory in which to hide bulky programs and that has a processor perhaps one-tenth the speed of a mainframe? Our particular experiences in doing protocol implementations for several different mainframes suggested that the slow bulky implementations are not intrinsic. Instead, they are brought about by a combination of several conquerable effects.

1) Although protocols are described in terms of layers, the particular layer structure chosen for description is not necessarily suitable for direct implementation. A naive implementation that places software modularity boundaries at the protocol layer boundaries can be extremely inefficient. The reason for the inefficiency is that in moving data, software modularity boundaries usually become the points where buffers and queues are inserted [3]. But the protocol layer boundaries are not necessarily the most effective points for buffering and queueing. A particular issue is that it is vital to minimize the number of times that data gets copied on the way from the application out to the wire and vice versa. (The operating system sometimes encourages extra copying too, but that is part of point 3), below.)

2) There are usually several ways to implement a protocol, all of which meet the specifications, but that can have radically different performance; the way that produces best performance for one application may be quite different from the way that produces best performance for another application. An implementation that tries to provide a general base for a wide variety of applications can perform much worse than one that is designed with one application in mind. This application variability of performance shows up strongly in the choice of data buffering strategies and in the choice of flow control strategies.

3) The current generation of operating systems is ill-equipped for integration of high-performance network protocols. Problems are often encountered in the form of a long time required to switch contexts, clumsy interprocess communication, or inadequate memory sharing. Good implementation of network protocols requires a very agile light-weight mechanism for coordination of intrinisically parallel activities—sending packets, receiving packets, sending packets at low levels as a result of receiving packets that require further processing at high levels, dallying in packet dispatch in hope that further processing will allow the piggybacking of responses at different levels into a single response packet, and so on. The various parallel activities of a network implementation are characterized by substantial sharing of both protocol state and packet data, so shared-variable communication is another essential feature. For an operating system to give really substantial support it needs to provide three forms of memory sharing: among operating-system-provided processes, between such processes and the operating system kernel, and between real and virtual memory areas. Often one or more of these three sharing mechanisms is missing or weak, so the protocol implementation is forced to spend time making copies.

One might summarize all three of these points by the single observation that current network protocol implementations, especially for high-speed local area networks, are quite early on the learning curve of this software area. Most experience so far is on large mainframes and with networks that operate at telephone line speeds. A particularly unexplored area is protocol implementations that work well with network speeds that span the four orders of magnitude from 1.2 kbits/s to 10 Mbits/s. One would expect that as experience is gained implementations will improve. One of the primary purposes of the PC network implementation was to take one or two steps higher on that learning curve.

In the remainder of this paper, we first describe what was implemented, and then discuss the organizing strategies that make the implementation interesting.

What Was Implemented

Fig. 1 shows the various protocols and drivers used within the PCIP software packages. PCIP divides naturally into three levels—the driver level, the transport level, and the application level. At the driver level are modules that manage different local area network hardware interfaces: the 3COM EtherLink Ethernet interface, the Proteon PRONET 10 Mbit/s token ring interface, and the RS-232 asynchronous serial line port.

The transport level has three major components. The Internet Protocol (IP) provides for packets originating on one network to be sent to a destination on another network. (In this paper the term "packet" describes the object delivered from one network-attached computer to another via the Internet Protocol.) The User Datagram Protocol (UDP) is a connectionless protocol intended for the transmission of a single uncontrolled datagram. The Transmission Control Protocol (TCP) provides a reliable full-duplex byte stream connection usually between a user process at one network node and a server process at another.

One application-level protocol, Remote Virtual Disk (RVD), is built directly on the IP layer. RVD is implemented as a device driver that allows one to read and write individual disk blocks on a remote machine as if they were on a local disk.

Several application-level protocols are built on UDP, each providing its own application-specialized error control. For example, the node name protocol takes a character string name for a node and consults a series of name servers to learn that node's 32 bit internet address, using UDP. The Trivial File Transfer Protocol (TFTP) is a lock-step file transfer protocol built on UDP, in which the file receiver must acknowledge each datagram before the sender may dispatch the next one. The Print File program permits a user to print a text file, using TFTP to transport the file to a printer server. The get time protocol obtains the time and date from a set of time and date servers.

The application programs that use TCP are the remote login protocol, named Telnet, and several information

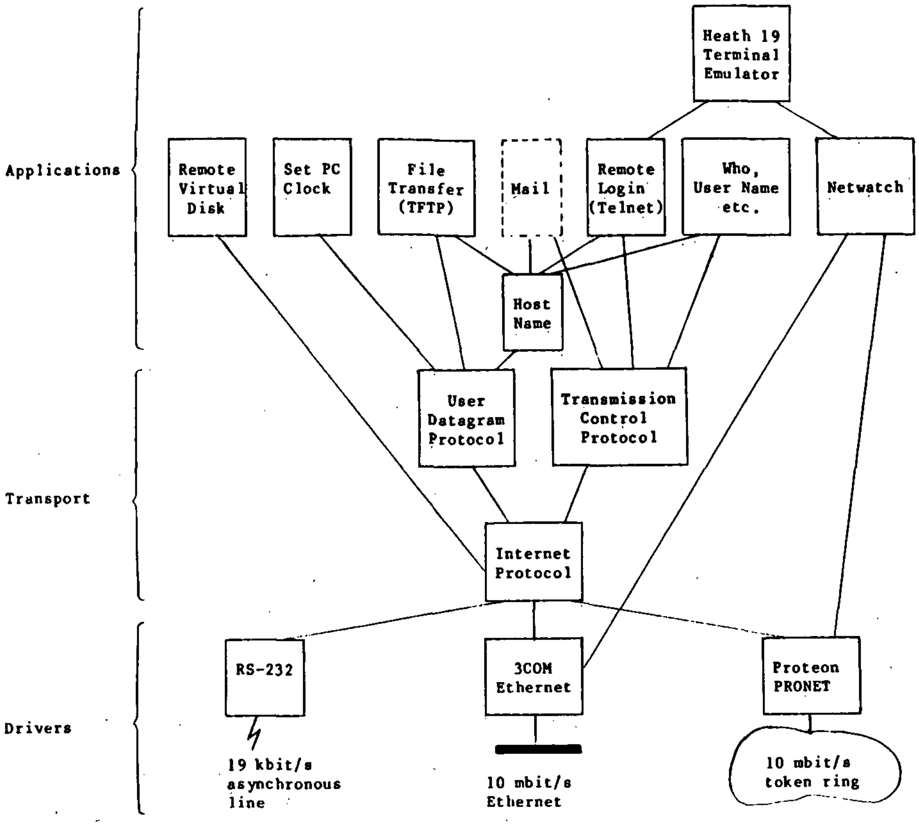


Fig. 1. PCIP Protocol hierarchy.

lookup protocols. In addition, some TCP-based mail facilities are currently being implemented. The Telnet program uses a Heath H19 terminal emulator in managing the keyboard and screen of the PC [4]. An unusual but very useful application program, Netwatch, built directly on the local area network drivers, allows monitoring of all or selected traffic on the local network. The ability to turn any PC into a network monitoring station at a moment's notice has been used to solve dozens of performance, configuration, and trouble isolation problems.

You Get More Than Remote Terminal Emulation

Although a remote login protocol is an important function, it is not by itself justification for a network implementation—if that were the only function obtained, one could use one of the many terminal emulator programs for the PC instead. The interest in a fuller protocol family implementation for a PC comes about when considering the range of services that become directly available for the PC user, and the ease of building new applications. Examples range from seemingly trivial ones to major work savers.

Among the apparently trivial features is the PC command that sets the PC system clock (date and time) by sending datagrams to several network servers [5]. This command is included in most of our PC users' automatic bootload batch files, where it eliminates the need for an extra battery-powered clock card. Only after this command became available did the date and time records kept in DOS floppy disk directories become reliable indicators of which version of a file one was looking at. Another remarkably useful command is one that obtains from any time-sharing system in the internet a list of currently logged-in users and identification information on any particular named user of that system. A similar command obtains directory information from the Arpanet Network Information Center. Another command sends a request for an echo response to any network node to allow trouble isolation. These tools, each not very important in itself, become part of an operation repertoire that makes the desktop computer much more useful than when it stands alone.

Probably the single most important tool is the file transfer protocol TFTP. TFTP provides the ability to move a file between the PC and any network-attached time-sharing system or file server. With TFTP, one can casually undertake quite complex operations. A typical use, such as the preparation of this paper, involves several authors each using a favorite editor on the PC to prepare individual contributions. Each moves contributions to a common directory on a central file server, so the others can look them over and provide comments and suggestions. One author moves all the paper fragments to a private PC, assembles them, runs them through a formatter and then sends them, again using TFTP, to a sophisticated laser printer server located elsewhere in the network. Because the network is not just local, but is seamlessly interconnected by the Arpanet to many other sites nationwide and worldwide, the authors and other facilities can be assembled from a geographically dispersed set. And because TFTP is an independent entity, it can be used as a subsystem in a more complex application, such as sending a text file to a manuscript preparation server, sending the result to a laser printer, and retrieving result log files, all in an under-the-cover application package that runs on the PC. Perhaps more important, if some file transfer mechanism had not been available, so that terminal emulation were the only means of communication, it would not have been easy to include stand-alone programs on the PC, such as text editors, as part of this process.

When added to this set of network tools, a remote login protocol becomes a migration and extension tool, since it makes functions that are missing on the PC, such as specialized languages or databases, easily available by allowing the PC user to attach to a time-sharing system anywhere in the network. The most prominent example of a function currently missing in our repertoire is electronic mail handling. While waiting for a mail handling package to be implemented for the PC, sending and receiving mail is accomplished by logging in to one of the large time-sharing systems. Another useful feature of implementing remote login as a network package rather than as a simple terminal emulator is that TFTP is available at the same time. This feature allows one to use any time-sharing system commands to locate, collect, or create files, and then send them immediately back to the PC. It also allows other users to move files to and from the PC at the same time it is being used for remote login.

The sum of these tools is greater than the individual parts because the application user can blend remote with local operations as the job requires. An application programmer finds even more flexibility since any service offered by any network site can be incorporated in an application program. It is exactly this ability to compose network functions in unanticipated ways that makes a complete network protocol implementation for a PC so powerful.

Remote Virtual Disk

A good example of an extended service possibility is our implementation of the Remote Virtual Disk protocol (RVD) for the PC. This protocol, locally developed at M.I.T., permits a machine to have access to disk storage which appears to be local, but which is in fact remotely located at a server across the network [6]. To accomplish this appearance, a device driver is written that, instead of reading and writing to a real disk, sends messages across the network to the RVD server which does the actual reads and writes.

There are a number of uses for the function provided by RVD. Most important, the disk made available through RVD can be shared, thus providing a mechanism for distribution of software, especially making a large library of tools available to a community. In this use, an RVD disk strongly resembles the virtual minidisk provided by the VM/370 operating system [7]. (Note, however, that if sharing is the primary goal, sharing at the physical disk block level is not as flexible as sharing at the logical file level. Remote file system protocols have been the subject of much research and development activity lately [8], [9].)

A simple but helpful use of the RVD disks is as an extension to the private disk storage of the individual machine. The economics of large and small disks is currently such that one has only modest price advantage over the other, but the functional advantage of RVD is threefold. First, any RVD disk can be available to every PC on the net, so in contrast to the permanently attached Winchester disk, the file stored on an RVD disk can still be reached if one's private PC is down, by walking down the hall and finding another network-attached PC. Second, since the RVD disks are actually partitions of centrally-located large disks, one can arrange for a central operations staff to make backup copies of the information stored on RVD disks. The need to make backup copies of information stored on private Winchester disks has proven to be one of the operational headaches of those devices; with RVD the headaches can be subcontracted to someone else. Third, the effective data rate of the RVD disk is comparable to a local hard disk and substantially better than that of a floppy disk. Large block transfers using RVD take place across the Ethernet at about 240 kbits/s.

The PC Environment

Development of a network implementation for the PC required that a number of choices be made, both in the development environment and in the programming environment. This section describes those choices.

The development environment, while it entailed difficult choices, did not involve any new ideas or breakthroughs. Programming was done on a microcomputer development system that runs on a nearby UNIX time-sharing system. That approach was used rather than doing the programming entirely on the PC because in 1981, when the choice was made, very little support software (editors, choice of compilers, library managers) was yet available to run on the PC. The programming was done in C language, with the choice again based primarily on the combination of compiler and assembler availability. It was apparent that some Assembly language programming would be required, and the only assembler that we could locate for the PC at the time was one that came as part of an integrated C compiler/assembler/loader package.

The programming environment used was the IBM DOS operating system [10]. This choice was easier than it might have appeared; all of the operating system alternatives provided very little support for the kinds of operations needed to do a network protocol implementation, so all required that support to be added. Thus the choice was made on predicted ubiquity, on which point DOS appeared strongest. The primary run-time facility added was a tasking and timer management package that permits as many parallel tasks as necessary to operate within a single address space. For simplicity, the tasking package runs each task to completion (either "block," awaiting a wakeup, or "yield," allowing other tasks to run) using a round robin scheduler.

The combination of the development environment and programming environment required one bootstrapping program to be constructed—an RS-232 port file-copying program for the PC that could take a file being pushed at it by UNIX and store it in the PC file system. The development environment on UNIX produced loaded, ready-to-run command files; the bootstrap provided a way of getting those command files into the PC for execution. The first real network program developed was one that implemented a standard file transfer protocol, and as soon as that program was operational the bootstrap was no longer needed [11].

PCIP Over Asynchronous Serial Ports

When the IBM PC was first announced there was no local area network interface available for it, but several manufacturers seemed intent on supplying them within a year or so. Rather than building a piece of special hardware that would be soon discarded, we opted to use the PC's asynchronous serial port as a temporary substitute. To connect the asynchronous serial port to an existing local area network, a token ring, we configured a Digital Equipment Corporation LSI-11 to contain both a token ring network interface and a small number of asynchronous serial ports. This LSI-11 came to be known as the PC-Gateway. The PC-Gateway was programmed to treat the set of asynchronous ports as a local network, and to act as a packet-forwarding gateway between that local network and the token ring. When the PC was ready to send a packet of data, it merely sent the packet as a sequence of 8 bit bytes over the asynchronous port. This approach made the combination of the asynchronous port driver, the port, the serial line, and the PC-Gateway a unit that could later be replaced by a local network driver and a network hardware interface.

There were two useful results from the PC-Gateway. First, it permitted substantial progress to be made in implementing and polishing the network code for the PC. When local network hardware did become available for the PC, the only software effort was to replace the asynchronous port driver with a network interface driver. Second, it turned out to be surprisingly useful and was not discarded when network interfaces arrived. Instead, dial-up modems were attached to unused asynchronous ports of the PC-Gateway to permit people who had PC's at home to connect to the network using telephone lines.

There was mixed success with the PC gateway using asynchronous lines. On a 9.6 kbit/s line, there was no major problem in performing either file transfers or using remote login, even with character-at-a-time remote echo. On a 1.2 kbit/s dial-up telephone line, file transfers were reasonably successful. (Sometimes the transmission time involved in sending a long packet over a 1.2 kbit/s line would cause the remote system to time-out and abort the file transfer. Eventually, most other implementations learned to be patient enough to tolerate telephone line transfers.) For remote login to time-sharing systems that work in character-at-a-time remote echo mode, each time the user typed a character, a packet in excess of 25 bytes was transmitted over the low-speed line. It was thus very easy for a fast typist to saturate the connection to the PC-Gateway, and echoing fell far behind the typist.

The problem of low-speed line saturation in one-character-per packet mode could have been overcome by employing data compression. Many of the bytes in each packet of a TCP connection are likely to be identical to those of the previous packet. An algorithm was discussed, but never implemented, to take advantage of this observation and transmit only the differences between the current packet and the previously transmitted packet. Compression was never implemented because the arrival of high-speed local area network interfaces reduced demand for remote login over 1.2 kbit/s lines. However, if the effort had been undertaken to increase performance on 1.2 kbit/s telephone lines, we believe that it would have been technically feasible.

One of the lessons learned from implementing the PC-Gateway was that a dial-up packet forwarder opens a new world of opportunities when compared with a dial-up terminal concentrator. When only terminal concentrator ports are available, files are usually transferred between mainframe computers and PC's using one of several embedded protocols such as KERMIT, developed at Columbia University [12]. One of the problems with such embedded protocols is that there are several different ones, so the PC and the mainframe may not have one in common. When that is the case, some staging process must be employed whereby the user first moves the file from its original site to one that implements the same protocol available on the PC, and then transfers the file over the asynchronous connection. In contrast, the PC-gateway allowed implementation of a standard network file transfer protocol (TFTP) on the PC itself, which made file transfer immediately usable with all the other network participants. But even more important, all other network services, such as node name resolution, time-of-day service, etc., become available to the PC, and the PC can respond to requests initiated by other systems, inquiring about presence, identity, existence of files, etc.

There is no reason why both terminal concentration and packet forwarding cannot be provided on a single port. Our advice to future implementors of network terminal concentrators is to provide an escape mechanism so that a PC can directly send and receive network packets carrying any protocol the PC finds useful. This escape can give the PC the opportunity to make fuller use of the network possibilities.

Tailoring the Implementation to the Environment

There are a few characteristics of desktop computer operation that are quite different from mainframe operation, and these characteristics affect the way in which the network is integrated with the system. The most important of these is that the desktop computer is often—perhaps usually—not "on the network." When not in use, a desktop computer is often powered off, perhaps to reduce the noise and heat in the office in which it is located. Even when powered on, one cannot expect the network software to be always in operation. Some desktop computer application software packages operate by taking over the entire machine, sometimes to prevent pirate copies of the program from being made and sometimes simply because they require every scrap of memory in order to perform usably.

Thus the software in the personal computer cannot expect to maintain a continuous record of the state of the network; instead it must be organized so that it can quickly discover whatever state it needs when it is called into operation. To cope with the "normally-off-the-network" paradigm of operation, the various PCIP programs do not attempt to retain any discovered network information at all for the use of the next program that may use the network. Because one has no idea what other application program may run between two network programs, the up-to-dateness and integrity of any state variable stored in primary memory is questionable, and it is safer to rediscover the network information rather than to depend on a stored value. Thus if one initiates a file transfer with another site, such facts as the round trip time to that site, its network address, and the Ethernet address of an intervening gateway are all discovered, used during the transfer, and then discarded. If the next command to be typed is another file transfer to the same site the listed facts are all rediscovered again. This approach, while perhaps seeming wasteful, actually costs quite little and has a very large payoff in improved reliability of the network software. In contrast with our experience with other network implementations that maintain network state continuously, in PCIP one almost never encounters the situation in which anomalous behavior (caused by recorded state getting out of step with real state) leads to a need to reboot the system or explicitly reinitialize the network code to get it working again. (However, all is not roses. Because there is no protection between supervisor and user in the PC, bugs in either the network code or in a user application can cause a system crash, requiring a reboot to recover. During application programming such crashes are fairly common, providing another reason why one cannot depend on maintaining network state records.)

Another aspect of this expectation of frequent detachment from the network is that the PC network implementation makes no attempt at all to maintain a table of (user oriented, character string) names of other nodes and their network addresses. Keeping such a table in step with the name tables in nodes that are always on-line (and which depend on that usual on-lineness in informing one another of changes) would be a major challenge. So instead the PC depends on the availability of node name translation services provided by many of the always-on-line network systems.

A related problem is that the network software must be able to discover quickly environment parameters (such as network addresses of nearby gateways and other servers or the number of the network to which it is attached) rather than expecting that the user types them in each time when a network program is used. To provide such environment parameters, the PCIP implementation used a trick: a piece of code is installed as a DOS device driver, but this piece of code does not actually control a real device. Instead, calls to read from this device cause the code to send back a stream of environment information, in a standard format. Every PCIP program knows how to interpret this stream, and thereby has a quick way of discovering the facts about the environment it needs. A customization program allows the application user to set up this pseudodevice driver. Using a pseudodevice driver provides this information much more rapidly than reading a file, and it is far easier to change as compared with the alternative of assembling the information in as constants of the programs. (The DOS 2.0 environment variable feature in principle provides an equally good way to do this job, but unfortunately the space allocated by DOS for environment variables was insufficient.)

Tailoring the Implementation to the Application

Perhaps the most interesting strategy used by the PCIP software to obtain good performance in a small machine is the tailoring of the network implementation to match the application that will use it. There are several examples of this tailoring that illustrate the idea. The primary examples are in the implementation of the end-to-end transport protocol, TCP. This implementation was designed to work optimally with only one application protocol, the "User Telnet" remote login protocol [13]. The idea of tailoring is that the knowledge that the only application is remote login should guide implementation decisions in the transport protocol.

Some of the decisions simply relate to how much standard TCP function to implement. The PC TCP can only originate connections; no provision was made for other nodes to make connections to the PC because the feature is not needed by User Telnet. Similarly, PC TCP can maintain only one connection at a time, because User Telnet requires only one connection. A substantial amount of table management code is thus unneeded.

TCP includes a sliding window for flow control. The PC TCP simply ignores the window values sent to it by the remote login server, because when it is used for remote login, the only data sent to the login service are those typed by a person at a keyboard, and that data rate is almost certain to be lower than the rate that the service can accept data. (If once in a great while the service falls behind so far that the typist gets ahead of the offered window, no loss of data occurs—the login service simply stops acknowledging the data, and the PC TCP has for error control a time out-resend strategy that retries until the service catches up.) The simplicity that results from ignoring windows makes the code both smaller and faster.

To minimize copying of data and space occupied by packet buffers, the TCP send function is tailored in another way with the knowledge that data come from a typist. Only one packet buffer is provided for output data, and this packet buffer is set up with certain fields, such as the source and destination addresses, precalculated, since they never change. When the user types a character, Telnet calls the TCP send function with the character as the argument, and send merely drops the character into the precalculated packet buffer, adjusts any remaining fields, and calls the local network driver with a pointer to the packet buffer. Because the output is to a high-speed local area network the network driver will complete the dispatch of the packet before returning to TCP. It is thus safe for TCP to assume that it now has control of, and can change the contents of, the output buffer. If the user types another character before the login service acknowledges the earlier one, Telnet calls TCP as usual, but TCP's send function simply slips this new character into the same packet buffer following the earlier character, and dispatches the packet containing, now, two characters. If the earlier packet is lost in transit (and thus no acknowledgment of it ever comes back from the service) this new two-character packet will act as the resend.

This technique of adding characters to the output packet buffer as they are typed has a limit of course; if the typist fills the packet buffer (500 characters, which allows at least 30 s of frantic typing) before the remote service acknowledges the first character typed the typist must be asked to stop; the TCP send function simply returns an error condition to Telnet when the single packet buffer is full, and Telnet notifies the typist to desist. This situation occurs very rarely in practice. Normally, the remote service receives a packet and sends back an acknowledgment of the oldest typed characters. The PC TCP, upon seeing that acknowledgment, adjusts the characters in the output packet buffer by sliding them back so that the first unacknowledged character is first in the output buffer. Usually, the acknowledgment is for all the outstanding data, and no copying occurs at all.

This whole collection of techniques of output buffer management reduces path length, buffer space, and packet copying, but all of them depend on the knowledge that the send function will be used in a particular way. If one tried to use this tailored TCP to send a file consisting of many large blocks of data, its performance would be very poor. It might overrun the file server because it ignores that server's flow control windows, leading to many unnecessary retransmissions. It could accept only one packet of data to be sent at a time because it has only one packet buffer, and it cannot reuse that buffer until acknowledgment comes from the other end that the receiver has accepted the data. There would be much time spent copying the large blocks of data from one end of the packet buffer to the other as acknowledgments came back. And finally, the implementor of the file transfer program would find that the TCP send interface accepts only one byte on each call, so sending a block of data would require an inefficient repeated call loop.

For data flowing to the PC, a completely different set of considerations holds. In this direction, the PC TCP implements flow control windows because it can be overrun by an active high-powered time-sharing system. However, there are still opportunities for tailoring the implementation.

The most serious problem with incoming data is not just that it arrives too fast, but that in the Arpanet some servers sometimes transmit a separate packet for each byte of data they send. Since the TCP window controls the number of outstanding bytes rather than the number of outstanding packets, the window does not prevent a flood of packets if the data is being sent in this very inefficient way. The problem shows up if the PC cannot keep up with the rate of arriving packets; fairly soon a packet gets missed and thus not acknowledged. The sending site eventually times out and resends, starting with the missed packet. The time-out shows up as a noticeable pause in the flow of data to the user's screen. The PC TCP required a special buffering scheme to deal with a large number of arriving small packets. Since running a complete terminal emulator is actually more time-consuming than processing incoming packets, the PC emulator is permitted to handle only a few bytes at a time before returning to the TCP level to see if more packets have come in. This strategy permits as much processor time as possible to be allocated to packet handling. (As described in the next section, the PC terminal emulator is invoked by an "upcall" from TCP, so limiting it is actually quite easy—TCP simply calls the emulator with an argument consisting of the number of characters it thinks the emulator can handle.)

This implicit flow control mechanism between the emulator and TCP replaces the more general explicit flow control system that would have to be implemented if TCP had been designed to cope with arbitrary client protocols including, for example, file transfer.

At least one more minor opportunity for tailoring exists in this direction. Since the customer application is remote login, it is a good bet that the largest quantity of data that will ever arrive in a single burst over a connection from the remote login service is one screen full, a predictably finite amount of data. Thus TCP input buffers and window size need be provided just for this amount and no more. If an ambitious server process aspires to send more than one screenful of data in a burst, the window mechanism acts as a throttle. In the most common case everything proceeds smoothly and optimally and the window is not a limit. In an unusual case performance may suffer but no data are lost.

Upcalls

The combination of the tasking package and the C language features of static storage and procedure variables are used extensively throughout the network implementation in a style of programming known locally as "upcall/downcall." (In some of the more recently developed window management systems, and the Pilot file system, the same style of programming is sometimes known as "callback" [14].) In this style of programming some tasks are waiting for events as "high" levels, for example, in application programs. When an event occurs the application program operates by calling "down" to lower level network implementation programs. This is the usual style of programming of operating systems. However, other tasks wait for signals at low levels, inside network driver programs, for example. When a signal starts them, perhaps because a packet has arrived, they operate by handling the packet operations at their level, and then calling "up" to higher levels of network protocol and eventually "up" to the application.

The denotation "up" and "down" can be misleading, because a call "up" can lead to a call "down" as part of its implementation. For example, the arrival of a packet may result in an upcall to dispose of the packet and during the upcall one or more downcalls to send acknowledgments, flow control messages, or an application-level response.

Fig. 2 illustrates in a simplified example the use of this organization in the implementation of the Telnet remote login protocol. In that figure, in the left column, the top level application program creates a parallel task (in the right column) to handle arriving packets using upcalls. The top level program proceeds to initialize static procedure variables in anticipation of upcalls at the several network protocol levels. The main task then concentrates on sending typed characters to the remote server. Meanwhile, in the right column, all packets coming from the remote server are noticed at a low level by the network driver, which calls upward, using the previously initialized tables of procedure variables, eventually reaching the screen display procedure of the terminal emulator. Although the actual programs are complicated by error conditions, the basic flow of control illustrated in this figure is complete and, relative to other implementations we have seen, quite simple [15].

```
MAIN TASK (DOWNCALL FOR               | RECEIVE TASK (UPCALL WHEN
        INITIALIZATION)               |           PACKET ARRIVES)
--------------------------------------+----------------------------------
Telnet:                               |
  fork(ether_rcv)                     | telnet_rcv(char):
  tcp_open(telnet_rcv)                |   display char on screen
  do always                           |   return
    data <-- read_keyboard            |
    tcp_send(data)                    |
    end                               |
--------------------------------------+----------------------------------
tcp_open(cproc):                      | tcp_rcv(packet):
  tcp_customer <-- cproc              |   for each character in pkt
  ether_open(tcp_rcv, tcp_type)       |     call tcp_customer(char)
  return                              |   return
--------------------------------------+----------------------------------
ether_open(proc, type):               | ether_rcv:
  customer(type) <-- proc             |   start ether receiver
  return                              |   do always
                                      |     block till packet arrives
                                      |     get arriving packet type
                                      |     cust_proc <-- customer
                                      |                 (packet_type)
                                      |     call cust_proc(packet)
                                      |     end
```

Fig. 2. PC/Telnet call organization. The downcall flow, on the left, initializes static procedure variables for later upcalls. The identifier "tcp-type" is a constant set to the protocol-specified value that distinguishes TCP packets from other kinds. When a packet arrives from the network, it activates the upcall flow on the right.

The upcall/downcall programming style, together with a tasking package that allows several tasks to operate within a single address space is the primary set of tools used to gain leverage against the third performance-draining effect mentioned earlier—that the current generation of operating systems does not provide agile lightweight support for the parallel operations that are required to run a network implementation. An upcall also provides a natural way for a network implementation layer to receive data from below and pass them up higher without having to copy them just to insure that they do not get deallocated by the lower level. Thus some leverage is also obtained against the first performance-draining effect—too much buffering at protocol layer boundaries. Another example of the simplifying effect of upcalls was mentioned in the previous section, which described their use to provide implicit flow control between TCP and Telnet.

Getting Around DOS

The implementation of the Remote Virtual Disk protocol for the PC was an interesting exercise. The DOS operating system has a provision for user-installed disk drivers, so there was an obvious place to integrate the RVD interface. However, the RVD driver is rather different from most drivers; since it implements a network protocol inside, it contains all the support tools we implemented for the other protocol packages, including our tasking scheduler and our timer manager. Since PC DOS is not designed to be reentrant, the driver cannot call on DOS for any services, so it must recreate any DOS functions it needs. The resulting exercise causes the implementer of RVD to stand on his head to get some things done, and produces a device driver for DOS with considerably more sophisticated operating system features than DOS itself.

There was one limitation imposed by DOS that we have not yet tried to circumvent. Since the network package for RVD was hidden inside what DOS thought was a disk driver, that network package was not available for use by other applications. Since only one driver can control the physical network interface at a time, no other network application could use RVD service. This limitation meant that, for example, one could not use the file transfer protocol to move a file to or from an RVD disk. Such transfers currently require a two-stage operation, moving the file via a disk physically at the local PC and copying it from there to, or from, the RVD disk. Removing this restriction could be done by adding an ad hoc communication path directly from the application to the RVD package, a path that a more flexible operating system might have provided.

Our experience with RVD clearly showed that the PC had enough power to support this kind of protocol, and that such a feature could be very helpful. Even with its limitations, RVD is in wide use in our laboratory. However, the limitations of DOS 2.0 increased the difficulty of this project, and reduced somewhat the value of the final service. Fortunately, this sort of limitation seems to be going away as the creators of operating systems expand their vision of the capabilities of a PC class machine. For example, in a UNIX-based operating system such as XENIX or PCIX, the restriction on RVD use by other network applications would not be necessary.[1]

On Size and Scale

While the CPU of the PC can access 1 Mbyte of memory, all of the PCIP packages can operate in a 128 kbyte configuration. (This small size was fortunate, because it happened that the available C compiler used a "small memory model," limiting one loaded program to 64 kbytes of code and 64 kbytes of data.) The individual packages are relatively small; combined they easily meet this constraint. Consider the decomposition of the code space of the file transfer package, TFTP:

TFTP user and server	7650 bytes
UDP	2812
IP	4614
Ethernet driver	5744
Network common library	1198
Timer and tasking package	2314
C run time support	7569
Total	31 901 bytes

The largest most complex package is the Telnet command. It uses TCP and UDP (for name resolution) and contains a TFTP server. This command consists of the modules above, plus

Telnet	7392 bytes
TCP	5946
Total	45 339 bytes

The size of Telnet includes the size of the screen manager as well as the protocol implementation. Notice that Telnet and TCP are individually the most complex modules implemented.

An interesting observation about the scale of a network package for a personal computer comes from examination of a typical package, the one that does file transfer. The implementation of TFTP user and server is done in three C language programs and one C language "include" file of common data structure definitions. That set of programs implements just the box labeled "Trivial File Transfer" in Fig. 1. These C programs together total about 1020 lines of code (excluding comments), of which about 450 lines implement the main stream of the protocol, 505 lines handle error conditions, and 65 were provided as aids for debugging. The 50 percent figure for handling error conditions in our experience is typical for network code that is intended to be reasonably robust. A similar fraction was noted by Clark in his implementation of the TFTP protocol in PL/I for the Multics system. Probably much more than half the intellectual effort of design and debugging went into that part of the code, since it tends to involve untangling of things that did not go right, rather than straightforwardly moving on to the next step of the protocol. The 1000 line figure for TFTP as a whole indicates that the overall size of network packages is well within the capability of a desktop computer.

The lesson to be drawn from all these numbers is that with proper system support, good organization, and attention to the client being supported, a network protocol package need not be a large module.

When we examine the performance of the programs, we find that the bottlenecks are not in the protocol implementations themselves, but in resources the applications utilize. The code was not written with great concern for performance because it was expected that the bottlenecks would be found outside of the protocol implementations. The low cost of context switching and few data copies allow fast transfer of data through the protocol layers. For instance, TFTP writing to a floppy disk frequently achieves an end-to-end useful data rate of 13 kbits/s, about the writing speed of the floppy disk. With a Winchester disk, TFTP can transfer data over the network at a rate of about 55 kbits/s, again about the writing speed (for small blocks) of the disk drive itself. When tests are done in which TFTP discards data as soon as it is received, network transfers run as fast as 110 kbits/s. Thus the bottlenecks in file transfer seem to be the disk systems, and improvements that we might make to the protocol implementation would not substantially alter the transfer rates achieved.

A second example is Telnet. Monitoring shows that it spends 50 percent of its processing time in the Heath H19 terminal emulator. Another 30 percent is spent idle, waiting for something to do. For a real performance breakthrough in Telnet, the terminal emulator would have to be improved, rather than the IP or TCP implementation. While some speed could be gained by small changes to the TCP implementation, the terminal emulator is the real bottleneck.

Conclusions

In the beginning of this paper, we identified three problems that can beset the implementor of network protocols.

1) The architected layer structure of the protocol can prove unsuitable as a structuring technique for the implementation.

2) An implementation that attempts to serve several clients may either be very complex or provide poor performance to some or all clients.

3) The operating system chosen may provide poor support for the needed program structure.

The impact of these problems is that a full implementation of a protocol suite tends to be sufficiently bulky and slow that a realization inside a personal computer seems impractical. We have shown to our satisfaction that this

[1] UNIX is a trademark of AT&T Bell Laboratories. XENIX is a trademark of MicroSoft Corporation. PCIX is a trademark of Interactive Systems Corporation.

need not be so. We produced a running and useful implementation that is consistent with the speed and size of an IBM PC by identifying and using techniques that directly combat the problems identified above.

To avoid the excessive interfacing code that results from classical layering we used an interface technique, upcalls, that put the asynchronous boundaries in the implementation only where they are needed. Subroutine calls, always more efficient than process switches, are used wherever possible.

To combat the high cost of generality we abandoned it wherever abandonment really seemed to pay off. Instead of producing a virtual circuit protocol that attempted good performance for all clients, we tailored the implementation to remote login. Compared to other implementations of more generality that we have examined, this code was substantially smaller and simpler to produce.

To solve the problem of an unsuitable operating system, we provided fragments of our own, as part of the network code. This kind of replacement is not always possible, but in this case it both proved the benefit of proper system support for protocols, and demonstrated the flexibility of the programming environment of the PC.

We feel very strongly that it is a good approach to produce implementations that are tailored to specific clients, as opposed to more general implementations. The drawback of this technique is that if several clients are to be supported, it is necessary to produce several different implementations of the support program, which produces unwelcome increases in maintenance costs. In other projects we have done this sort of multiple implementation, and do not feel that the development effort is substantial. Many parts of the implementation, such as the protocol state machine, can be reused. To help control the maintenance cost, we are now exploring different modularity techniques in which the protocol state machine for a layer is implemented as a general module, while the data flow paths are supplied by each client, using a standard interface.

Acknowledgment

The first implementation of TFTP was accomplished by K. Wright and the initial implementation of Telnet was done by L. Konopelski. D. Bridgham wrote the terminal emulator used in Telnet. C. Terman kindly supplied the C language development system. Several early users, including especially F. Corbató and R. Iannucci, acted as uncomplaining guinea pigs while the network code was being debugged. Finally, one of the anonymous referees of this paper provided extensive comments of unusual depth and insight.

References

[1] *Internet Protocol Transition Workbook*, SRI Int. Network Inform. Center, Menlo Park, CA, Mar. 1982.

[2] *Internet Protocol Implementation Guide*, SRI Int. Network Inform. Center, Menlo Park, CA, Aug. 1982.

[3] G. H. Cooper, "An argument for soft layering of protocols," S. M. thesis, Dep. Elec. Eng. and Comput. Sci., Massachusetts Inst. Technol., May, 1983; see also M.I.T. Lab. Comput. Sci. Tech. Rep. TR-300, May, 1983.

[4] *Heathkit Manual for the Video Terminal, Model H19*, Heath Co., Benton Harbor, MI, 1979.

[5] J. H. Saltzer, *PC/IP User's Guide*, M.I.T. Lab. Comput. Sci., Dec. 1984.

[6] M. Greenwald, "Remote virtual disk protocol specifications," Massachusetts Inst. Technol. Lab. Comput. Sci., Cambridge, MA, Tech. Memo., in preparation.

[7] L. H. Seawright, and R. A. MacKinnon, "VM/370—A study of multiplicity and usefulness," *IBM Syst. J.*, vol. 19, no. 1, pp. 4–17, 1979.

[8] J. Laselle *et al.*, *EtherShare User's Guide*, 3COM Corp., Mountain View, CA, July 1983.

[9] B. C. Goldstein *et al.*, "Directions in cooperative processing between workstations and hosts," *IBM Syst. J.*, vol. 23, no. 3 pp. 236–244, 1984.

[10] *Microsoft Disk Operating System*, Version 2.0, IBM Corp., Boca Raton, FL, Jan. 1983.

[11] Karl D. Wright, "A file transfer program for a personal computer," S. B. thesis, Dep. Elec. Eng. and Comput. Sci., Massachusetts Inst. Technol., Apr. 1982; see also M.I.T. Lab. Comput. Sci. Tech. Memo. TM-217, Apr. 1982.

[12] F. DaCruz and B. Catchings, "KERMIT: A file-transfer protocol for universities," *BYTE*, vol. 9, no. 6, pp. 255–278, June 1984.

[13] Louis J. Konopelski, "Implementing internet remote login on a personal computer," S. B. thesis, Dep. of Elec. Eng. and Comput. Sci., Massachusetts Inst. of Tech., Dec. 1982; see also M.I.T. Lab. Comput. Sci. Tech. Memo. TM-233, Dec. 1982.

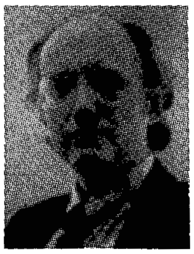
[14] L. R. Reid and P. L. Karlton, "A file system supporting cooperation between programs," *ACM Operat. Syst. Rev.* vol. 17, no. 5, pp. 20–29, Oct. 1983.

[15] J. L. Romkey, *IBM PC Network Programmer's Manual*, M.I.T. Lab. Comput. Sci., Dec. 1984.

Jerome H. Saltzer (M'69–SM'75–F'83) was born in Nampa, ID, on October 9, 1939. He received the S.B., S.M., and Sc.D. degrees in electrical engineering from the Massachusetts Institute of Technology, Cambridge, in 1961, 1963, and 1966, respectively.

Since 1966, he has been a faculty member of the Department of Electrical Engineering and Computer Science at M.I.T., where he has been active in the formulation of the undergraduate curriculum in Computer Science, including development of the core subject on the engineering of computer systems. At the M.I.T. Laboratory for Computer Science he participated in the refinement of the Compatible Time-Sharing System (CTSS) and then was involved in all aspects of the design and implementation of the Multiplexed Information and Computing Service (Multics). More recently, his research activities have involved the design of a token-passing ring local network, exploration of the problems of high-performance protocol implementations, and interenterprise connection. He is currently Technical Director of M.I.T. Project Athena, using networking desktop computers to improve undergraduate science and engineering education.

Prof. Saltzer is a member of the Association for Computing Machinery, the American Association for the Advancement of Science, Sigma Xi, Eta Kappa Nu, and Tau Beta Pi.



David D. Clark (M'66) received the B.S.E.E. degree with distinction from Swarthmore College, Swarthmore, PA, in 1966, the S.M. and E.E. degrees in 1968, and the Ph.D. degree in 1973, all from the Massachusetts Institute of Technology, Cambridge.

Since 1979 he has been associated with the Laboratory for Computer Science, M.I.T., where he is currently a Senior Research Scientist in the Distributed Computer Systems group. His first activities at M.I.T. were a variety of projects related to the development of the Multics operating system, including design of an I/O system and a programming language for system implementation. He has been a participant in the development of the Arpanet during the last ten years, and is currently chief protocol architect for the DARPA Internet project, which is developing standards for Department of Defense networking. His research efforts are in the development of efficient network protocols and computer operating systems suitable for use with high-speed local area networks.

Dr. Clark is a member of the Association for Computing Machinery and Sigma Xi.

John L. Romkey is currently a senior at the Massachusetts Institute of Technology, Cambridge, in the Department of Electrical Engineering and Computer Science. He expects to receive the B.S. degree in June of 1985.

He began working on a TCP/IP implementation with Prof. Jerome Saltzer in January 1982, and is currently working on a distributed mail system with Dr. David Clark.

Wayne C. Gramlich received the B.S. degree in electrical engineering and mathematics, with honors, in 1979, and the M.S. degree in electrical engineering with the computer engineering option, both from Carnegie-Mellon University, Pittsburgh, PA.

In 1979, he entered the graduate school at the Massachusetts Institute of Technology, Cambridge, working towards the Ph.D. degree in electrical engineering and computer science. His work has been in the areas of computer networks, personal computers, and distributed systems. His unfinished Ph.D. dissertation is entitled "Checkpoint Debugging." He is currently working for Sun Microsystems, Mountain View, CA, in the area of programming environments.