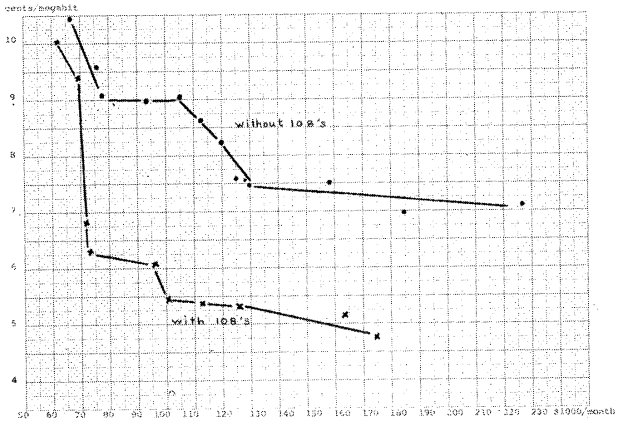


Figure 6

the effect of eliminating the 108 kilobit line option (which is not a standard AT&T offering), the cost per megabit of transmitted data is plotted against the total monthly line cost in Figure 6 for low cost networks designed with and without this option. Each point in this figure represents a feasible network. The points are connected by straight lines for visual convenience.

Additional investigations are presently under way to better understand the relationship between cost, delay and throughput, and the effect of the number of nodes on these parameters. Furthermore, alternative routing schemes will be considered as well as the cost-throughput tradeoffs that can be obtained by increasing the number of buffers at appropriate nodes.

## REFERENCES

1 L KLEINROCK
*Models for computer networks*
Proceedings of the International Conference on Communications pp 21.9–21.16 June 1969

2 L KLEINROCK
*Analytic and simulation methods in computer network design.*
See paper this conference

3 K STEIGLITZ P WEINER D KLEITMAN
*Design of minimum cost survivable networks*
IEEE Transactions on Circuit Theory 1970

4 B ROTHFARB M GOLDSTEIN
Unpublished work

5 H FRANK B ROTHFARB D KLEITMAN K STEIGLITZ
*Design of economical offshore natural gas pipeline networks*
Office of Emergency Preparedness Report No R-1 Washington D C January 1969

6 S LIN
*Computer solutions of the traveling salesman problem*
Bell System Tech Journal Vol 44 No 10 pp 2245–2269 December 1965

7 H FRANK I T FRISCH
*Communication, transmission, and transportation networks*
Addison–Wesley 1971

# HOST-HOST communication protocol in the ARPA network*

*by* C. STEPHEN CARR

*University of Utah*
Salt Lake City, Utah

and

STEPHEN D. CROCKER and VINTON G. CERF

*University of California*
Los Angeles, California

## INTRODUCTION

The Advanced Research Projects Agency (ARPA) Computer Netwok (hereafter referred to as the "ARPA network") is one of the most ambitious computer networks attempted to date.[1] The types of machines and operating systems involved in the network vary widely. For example, the computers at the first four sites are an XDS 940 (Stanford Research Institute), an IBM 360/75 (University of California, Santa Barbara), an XDS SIGMA-7 (University of California, Los Angeles), and a DEC PDP-10 (University of Utah). The only commonality among the network membership is the use of highly interactive time-sharing systems; but, of course, these are all different in external appearance and implementation. Furthermore, no one node is in control of the network. This has insured generality and reliability but complicates the software.

Of the networks which have reached the operational phase and been reported in the literature, none have involved the variety of computers and operating systems found in the ARPA network. For example, the Carnegie-Mellon, Princeton, IBM network consists of 360/67's with identical software.[2] Load sharing among identical batch machines was commonplace at North American Rockwell Corporation in the early 1960's. Therefore, the implementers of the present network have been only slightly influenced by earlier network attempts.

However, early time-sharing studies at the University of California at Berkeley, MIT, Lincoln Laboratory, and System Development Corporation (all ARPA sponsored) have had considerable influence on the design of the network. In some sense, the ARPA network of time-shared computers is a natural extension of earlier time-sharing concepts.

The network is seen as a set of data entry and exit points into which individual computers insert messages destined for another (or the same) computer, and from which such messages emerge. The format of such messages and the operation of the network was specified by the network contractor (BB&N) and it became the responsibility of representatives of the various computer sites to impose such additional constraints and provide such protocol as necessary for users at one site to use resources at foreign sites. This paper details the decisions that have been made and the considerations behind these decisions.

Several people deserve acknowledgment in this effort. J. Rulifson and W. Duvall of SRI participated in the early design effort of the protocol and in the discussions of NIL. G. Deloche of Thomson-CSF participated in the design effort while he was at UCLA and provided considerable documentation. J. Curry of Utah and P. Rovner of Lincoln Laboratory reviewed the early design and NIL. W. Crowther of Bolt, Beranek and Newman contributed the idea of a virtual net. The BB&N staff provided substantial assistance and guidance while delivering the network.

We have found that, in the process of connecting machines and operating systems together, a great deal of rapport has been established between personnel at

*This research was sponsored by the Advanced Research Projects Agency, Department of Defense, under contracts AF30(602)-4277 and DAHC15-69-C-0285.

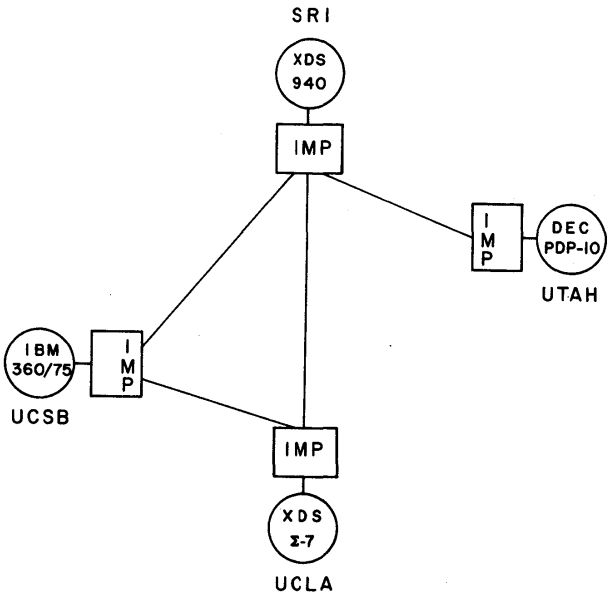


Figure 1—Initial network configuration

the various network node sites. The resulting mixture of ideas, discussions, disagreements, and resolutions has been highly refreshing and beneficial to all involved, and we regard the human interaction as a valuable by-product of the main effort.

## THE NETWORK AS SEEN BY THE HOSTS

Before going on to discuss operating system communication protocol, some definitions are needed.

A *HOST* is a computer system which is part of the network.

An *IMP* (Interface Message Processor) is a Honeywell DDP–516 computer which interfaces with up to four HOSTs at a particular site, and allows HOSTs access into the network. The configuration of the initial four-HOST network is given in Figure 1. The IMPs form a store-and-forward communications network. A companion paper in these proceedings covers the IMPs in some detail.[3]

A *message* is a bit stream less than 8096 bits long which is given to an IMP by a HOST for transmission to another HOST. The first 32 bits of the message are the *leader*. The leader contains the following information:

(a) HOST
(b) Message type
(c) Flags
(d) Link number

When a message is transmitted from a HOST to its IMP, the HOST field of the leader names the receiving HOST. When the message arrives at the receiving HOST, the HOST field names the sending HOST.

Only two message types are of concern in this paper. Regular messages are generated by a HOST and sent to its IMP for transmission to a foreign HOST. The other message type of interest is a RFNM (Request-for-Next-Message). RFNMs are explained in conjunction with links.

The flag field of the leader controls special cases not of concern here.

The link number identifies over which of 256 logical paths (links) between the sending HOST and the receiving HOST the message will be sent. Each link is unidirectional and is controlled by the network so that no more than one message at a time may be sent over it. This control is implemented using RFNM messages. After a sending HOST has sent a message to a receiving HOST over a particular link, the sending HOST is prohibited from sending another message over that same link until the sending HOST receives a RFNM. The RFNM is generated by the IMP connected to the receiving HOST, and the RFNM is sent back to the sending HOST after the message has entered the receiving HOST. It is important to remember that there are 256 links in each direction and that no relationship among these is imposed by the network.

The purpose of the link and RFNM mechanism is to prohibit individual users from overloading an IMP or a HOST. Implicit in this purpose is the assumption that a user does not use multiple links to achieve a wide band, and to a large extent the HOST–HOST protocol cooperates with this assumption. An even more basic assumption, of course, is that the network's load comes from some users transmitting sequences of messages rather than many users transmitting single messages coincidently.

In order to delimit the length of the message, and to make it easier for HOSTs of differing word lengths to communicate, the following formatting procedure is used. When a HOST prepares a message for output, it creates a 32-bit leader. Following the leader is a binary string, called *marking*, consisting of an arbitrary number of zeroes, followed by a one. Marking makes it possible for the sending HOST to synchronize the beginning of the text of a message with its word boundaries. When the last bit of a message has entered an IMP, the hardware interface between the IMP and HOST appends a one followed by enough zeroes to make the message length a multiple of 16 bits. These appended bits are called *padding*. Except for the marking and padding, no limitations are placed on the text of a message. Figure 2 shows a typical message sent by a 24-bit machine.

## DESIGN CONCEPTS

The computers participating in the network are alike in two important respects: each supports research inde-

pendent of the network, and each is under the discipline of a time-sharing system. These facts contributed to the following design philosophy.

First, because the computers in the network have independent purposes, it is necessary to preserve decentralized administrative control of the various computers. Since all of the time-sharing supervisors possess elaborate and definite accounting and resource allocation mechanisms, we arranged matters so that these mechanisms would control the load due to the network in the same way they control locally generated load.

Second, because the computers are all operated under time-sharing disciplines, it seemed desirable to facilitate basic interactive mechanisms.

Third, because this network is used by experienced programmers it was imperative to provide the widest latitude in using the network. Restrictions concerning character sets, programming languages, etc., would not be tolerated and we avoided such restrictions.

Fourth, again because the network is used by experienced programmers, it was felt necessary to leave the design open-ended. We expect that conventions will arise from time to time as experience is gained, but we felt constrained not to impose them arbitrarily.

Fifth, in order to make network participation comfortable, or in some cases, feasible, the software interface to the network should require minimal surgery on the HOST operating system.

Finally, we accepted the assumption stated above that network use consists of prolonged conversations instead of one-shot requests.

Those considerations led to the notions of connections, a Network Control Program, a control link, control commands, sockets, and virtual nets.

A *connection* is an extension of a link. A connection connects two processes so that output from one process is input to the other. Connections are simplex, so two connections are needed if two processes are to converse in both directions.

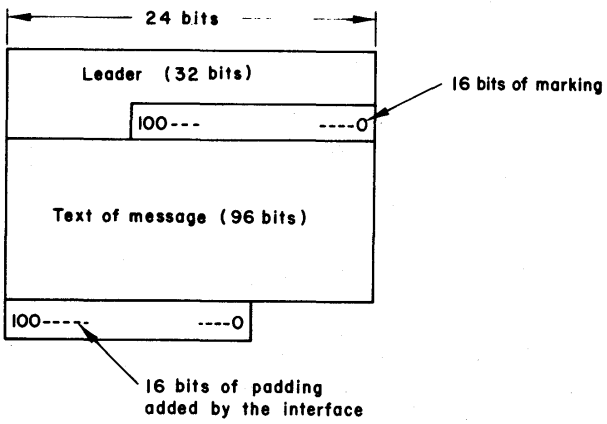


Figure 2—A typical message from a 24-bit machine

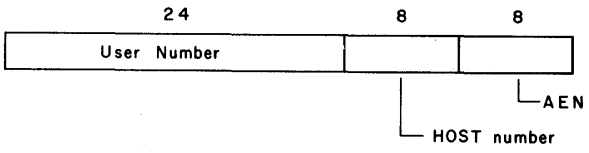


Figure 3—A typical socket

Processes within a HOST communicate with the network through a *Network Control Program* (NCP). In most HOSTs, the NCP will be part of the executive, so that processes will use system calls to communicate with it. The primary function of the NCP is to establish connections, break connections, switch connections, and control flow.

In order to accomplish its tasks, a NCP in one HOST must communicate with a NCP in another HOST. To this end, a particular link between each pair of HOSTs has been designated as the *control link*. Messages received over the control link are always interpreted by the NCP as a sequence of one or more *control commands*. As an example, one of the kinds of control commands is used to assign a link and initiate a connection, while another kind carries notification that a connection has been terminated. A partial sketch of the syntax and semantics of control commands is given in the next section.

A major issue is how to refer to processes in a foreign HOST. Each HOST has some internal naming scheme, but these various schemes often are incompatible. Since it is not practical to impose a common internal process naming scheme, an intermediate name space was created with a separate portion of the name space given to each HOST. It is left to each HOST to map internal process identifiers into its name space.

The elements of the name space are called *sockets*. A socket forms one end of a connection, and a connection is fully specified by a pair of sockets. A socket is specified by the concatenation of three numbers:

(a) a user number (24 bits)
(b) a HOST number (8 bits)
(c) AEN (8 bits)

A typical socket is illustrated in Figure 3.

Each HOST is assigned all sockets in the name space which have field (b) equal to the HOST's own identification.

A socket is either a *receive socket* or a *send socket*, and is so marked by the low-order bit of the AEN (0 = receive, 1 = send). The other seven bits of the

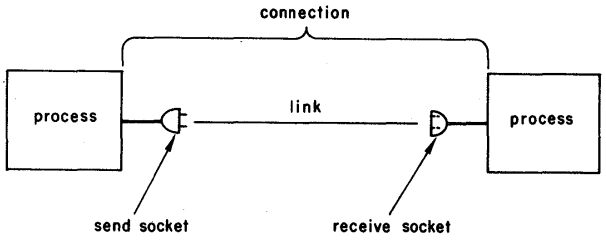


Figure 4—The relationship between sockets and processes

AEN simply provide a sizable population of sockets for each user number at each HOST. (AEN stands for "another eight-bit number".)

Each user is assigned a 24-bit user number which uniquely identifies him throughout the network. Generally this will be the 8-bit HOST number of his home HOST, followed by 16 bits which uniquely identify him at that HOST. Provision can also be made for a user to have a user number not keyed to a particular HOST, an arrangement desirable for mobile users who might have no home HOST or more than one home HOST. This 24-bit user number is then used in the following manner. When a user signs onto a HOST, his user number is looked up. Thereafter, each process the user creates is tagged with his user number. When the user signs onto a foreign HOST via the network, his same user number is used to tag processes he creates in that HOST. The foreign HOST obtains the user number either by consulting a table at login time, as the home HOST does, or by noticing the identification of the caller. The effect of propagating the user's number is that each user creates his own *virtual net* consisting of processes he has created. This virtual net may span an arbitrary number of HOSTs. It will thus be easy for a user to connect his processes in arbitrary ways, while still permitting him to connect his processes with those in other virtual nets.

The relationship between sockets and processes is now describable (see Figure 4). For each user number at each HOST, there are 128 send sockets and 128 receive sockets. A process may request from the local NCP the use of any one of the sockets with the same user number; the request is granted if the socket is not otherwise in use. The key observation here is that a socket requested by a process cannot already be in use unless it is by some other process within the same virtual net, and such a process is controlled by the same user.

An unusual aspect of the HOST–HOST protocol is that a process may switch its end of a connection from one socket to another. The new socket may be in any virtual net and at any HOST, and the process may initiate a switch either at the time the connection is being established, or later. The most general forms of switching entail quite complex implementation, and are not germane to the rest of this paper, so only a limited form will be explained. This limited form of switching provides only that a process may substitute one socket for another while establishing a connection. The new socket must have the same user number and HOST number, and the connection is still established to the same process. This form of switching is thus only a way of relabelling a socket, for no change in the routing of messages takes place. In the next section we document the system calls and control commands; in the section after next, we consider how login might be implemented.

## SYSTEM CALLS AND CONTROL COMMANDS

Here we sketch the mechanics of establishing, switching and breaking a connection. As noted above, the NCP interacts with user processes via system calls and with other NCPs via control commands. We therefore begin with a partial description of system calls and control commands.

System calls will vary from one operating system to another, so the following description is only suggestive. We assume here that a process has several input–output paths which we will call *ports*. Each port may be connected to a sequential I/O device, and while connected, transmits information in only one direction. We further assume that the process is blocked (dismissed, slept) while transmission proceeds. The following is the list of system calls:

Init ⟨port⟩, ⟨AEN 1⟩, ⟨AEN 2⟩, ⟨foreign socket⟩

where ⟨port⟩ is part of the process issuing the Init

and ⟨AEN 1⟩, ⟨AEN 2⟩ are 8-bit AEN's (see Figure 3)

⟨foreign socket⟩ is the 40-bit socket name of the distant end of the connection.

The first AEN is used to initiate the connection; the second is used while the connection exists.

The low-order bits of ⟨AEN 1⟩ and ⟨AEN 2⟩ must agree, and these must be the complement of the low-order bit of ⟨foreign socket⟩.

The NCP concatenates ⟨AEN 1⟩ and ⟨AEN 2⟩ each with the user number of the process and the HOST number to form 40-bit sockets.

It then sends a Request for Connection (RFC) control command to the distant NCP. When the distant NCP responds positively, the connection is established and

the process is unblocked. If the distant NCP responds negatively, the local NCP unblocks the requesting process, but informs it that the system call has failed.

Listen ⟨port⟩, ⟨AEN 1⟩

where ⟨port⟩ and ⟨AEN 1⟩ are as above.

The NCP retains the ports and ⟨AEN 1⟩ and blocks the process. When an RFC control command arrives naming the local socket, the process is unblocked and notified that a foreign process is calling.

Accept ⟨AEN 2⟩

After a listen has been satisfied, the process may either refuse the call or accept it and switch it to another socket. To accept the call, the process issues the Accept system call. The NCP then sends back an RFC control command.

Close ⟨port⟩

After establishing a connection, a process issues a Close to break the connection. The Close is also issued after a Listen to refuse a call.

Transmit ⟨port⟩, ⟨addr⟩

If ⟨port⟩ is attached to a send socket, ⟨addr⟩ points to a message to be sent. This message is preceded by its length in bits.

If ⟨port⟩ is attached to a receive socket, a message is stored at ⟨addr⟩. The length of the message is stored first.

*Control commands*

A vocabulary of control commands has been defined for communication between Network Control Programs. Each control command consists of an 8-bit operation code to indicate its function, followed by some parameters. The number and format of parameters is fixed for each operation code. A sequence of control commands destined for a particular HOST can be packed into a single control message.

RFC ⟨my socket 1⟩, ⟨my socket 2⟩,
⟨your socket⟩, (⟨link⟩)

This command is sent because a process has executed either an Init system call or an Accept system call. A link is assigned by the prospective receiver, so it is omitted if ⟨my socket 1⟩ is a send socket.

There is distinct advantage in using the same commands both to initiate a connection (Init) and to accept a call (Accept). If the responding command were different from the initiating command, then two processes could call each other and become blocked waiting for each other to respond. With this scheme no deadlock occurs and it provides a more compact way to connect a set of processes.

CLS ⟨my socket⟩, ⟨your socket⟩

The specified connection is terminated

CEASE ⟨link⟩

When the receiving process does not consume its input as fast as it arrives, the buffer space in the receiving HOST is used to queue the waiting messages. Since only limited space is generally available, the receiving HOST may need to inhibit the sending HOST from sending any more messages over the offending connection. When the sending HOST receives this command, it may block the process generating the messages.

RESUME ⟨link⟩

This command is also sent from the receiving HOST to the sending HOST and negates a previous CEASE.

## LOGGING IN

We assume that within each HOST there is always a process in execution which listens to login requests. We call this process the *logger*, and it is part of a special virtual net whose user number is zero. The logger is programmed to listen to calls on socket number 0. Upon receiving a call, the logger switches it to a higher (even) numbered socket, and returns a call to the socket numbered one less than the send socket originally calling. In this fashion, the logger can initiate 127 conversations.

To illustrate, assume a user whose identification is X′010005′ (user number 5 at UCLA) signs into UCLA, starts up one of his programs, and this program wants to start a process at SRI. No process at SRI except the logger is currently willing to listen to our user, so he executes

Init, ⟨port⟩ = 1, ⟨AEN 1⟩ = 7,
⟨AEN 2⟩ = 7,
⟨foreign socket⟩ = 0.

His process is blocked, and the NCP at UCLA sends

RFC ⟨my socket 1⟩ = X′0100050107′,
⟨my socket 2⟩ = X′0100050107′,
⟨your socket⟩ = X′0000000200′

The logger at SRI is notified when this message is received, because it has previously executed

Listen ⟨port⟨ = 9, ⟨AEN 1⟩ = 0.

(i) .LOGIN(cr)

(ii) .R TELNET(cr)

(iii) ESCAPE CHARACTER IS #(cr)

(iv) CONNECT TO SRI(cr)

(v) @ENTER CARR.(cr)

(vi) @CAL.(cr)

(vii) CAL AT YOUR SERVICE(cr)

(viii) >READ FILE FROM NETWRK.(cr)

(ix) #NETWRK:←DSK:MYFILE.CAL(cr)

Figure 5—A typical TELNET dialog
Underlined characters are those typed by the user

The logger then executes

Accept ⟨AEN 2⟩ = 88.

In response to the Accept, the SRI NCP sends

RFC ⟨my socket 1⟩ = X′0000000200′
⟨my socket 2⟩ = X′0000000258′
⟨your socket⟩ = X′0100050107′
⟨link⟩ = 37

where the link has been chosen from the set of available links. The SRI logger then executes

Init ⟨port⟩ = 10
⟨AEN 1⟩ = 89, ⟨AEN 2⟩ = 89,
⟨foreign socket⟩ = X′0100050106′

which causes the NCP to send

RFC ⟨my socket 1⟩ = X′0000000259′
⟨my socket 2⟩ = X′0000000259′
⟨your socket⟩ = X′0100050106′

The process at UCLA is unblocked and notified of the successful Init. Because the SRI logger always initiates a connection to the AEN one less than it has just been connected to, the UCLA process then executes

Listen ⟨port⟩ = 11
⟨AEN 1⟩ = 6

and when unblocked,

Accept ⟨AEN 2⟩ = 6.

When these transactions are complete, the UCLA process is doubly connected to the logger at SRI. The logger will then interrogate the UCLA process, and if satisfied, create a new process at SRI. This new process will be tagged with the user number X′010005′, and both connections will be switched to the new process. In this case, switching the connections to the new process corresponds to "passing the console down" in many time-sharing systems.

## USER LEVEL SOFTWARE

At the user level, subroutines which manage data buffers and format input destined for other HOSTs are provided. It is not mandatory that the user use such subroutines, since the user has access to the network system calls in his monitor.

In addition to user programming access, it is desirable to have a subsystem program at each HOST which makes the network immediately accessible from a teletype-like device without special programming. Subsystems are commonly used system components such as text editors, compilers and interpreters. An example of a network-related subsystem is *TELNET*, which will allow users at the University of Utah to connect to Stanford Research Institute and appear as regular terminal users. It is expected that more sophisticated subsystems will be developed in time, but this basic one will render the early network immediately useful.

A user at the University of Utah (UTAH) is sitting at a teletype dialed into the University's PDP–10/50 time-sharing system. He wishes to operate the Conversational Algebraic Language (CAL) subsystem on the XDS–940 at Stanford Research Institute (SRI) in Menlo Park, California. A typical TELNET dialog is illustrated in Figure 5. The meaning of each line of dialog is discussed here.

(i) The user signs in at UTAH.
(ii) The PDP–10 run command starts up the TELNET subsystem at the user's HOST.
(iii) The user identifies a break character which causes any message following the break to be

interpreted locally rather than being sent on to the foreign HOST.

(iv) The TELNET subsystem will make the appropriate system calls to establish a pair of connections to the SRI logger. The connections will be established only if SRI accepts another foreign user.

The UTAH user is now in the pre-logged-in state at SRI. This is analogous to the standard teletype user's state after dialing into a computer and making a connection but before typing anything.

(v) The user signs in to SRI with a standard login command.

Characters typed on the user's teletype are transmitted unaltered through the PDP–10 (user HOST) and on to the 940 (serving HOST). The PDP–10 TELNET subsystem will have automatically switched to full-duplex, character-by-character transmission, since this is required by SRI's 940. Full duplex operation is allowed for by the PDP–10, though not used by most Digital Equipment Corporation subsystems.

(vi) and (vii) The 940 subsystem, CAL, is started. At this point, the user wishes to load a CAL file into the 940 CAL subsystem from the file system on his local PDP–10.

(viii) CAL is instructed to establish a connection to UTAH in order to receive the file. "NETWRK" is a predefined 940 name similar in nature to "PAPER TAPE" or "TELETYPE".

(ix) Finally, the user types the break character ( #) followed by a command to his PDP–10 TELNET program, which sends the desired file to SRI from Utah on the connection just established for this purpose. The user's next statement is in CAL again.

The TELNET subsystem coding should be minimal for it is essentially a shell program built over the network system calls. It effectively establishes a shunt in the user HOST between the remote user and a distant serving HOST.

Given the basic system primitives, the TELNET subsystem at the user HOST and a manual for the serving HOST, the network can be profitably employed by remote users today.

## HIGHER LEVEL PROTOCOL

The network poses special problems where a high degree of interaction is required between the user and a particular subsystem on a foreign HOST. These problems arise due to heterogeneous consoles, local operating system overhead, and network transmission delays. Unless we use special strategies it may be difficult or even impossible for a distant user to make use of the more sophisticated subsystems offered. While these difficulties are especially severe in the area of graphics, problems may arise even for teletype interaction. For example, suppose that a foreign subsystem is designed for teletype consoles connected by telephone, and then this subsystem becomes available to network users. This subsystem might have the following characteristics.

1. Except for echoing and correction of mistyping, no action is taken until a carriage return is typed.
2. All characters except " ↑ ", "←" and carriage return are echoed as the character typed.
3. ← causes deletion of the immediately preceding accepted character, and is echoed as that character.
4. ↑ causes all previously typed characters to be ignored. A carriage return and line feed are echoed.
5. A carriage return is echoed as a carriage return followed by a line feed.

If each character typed is sent in its own message, then the characters

H E L L O ← ← P c.r.

cause nine messages in each direction. Furthermore, each character is handled by a user level program in the local HOST before being sent to the foreign HOST.

Now it is clear that if this particular example were important, we would quickly implement rules 1 to 5 in a local HOST program and send only complete lines to the foreign HOST. If the foreign HOST program could not be modified so as to not generate echoes, then the local program could not only echo properly, it could also throw away the later echoes from the foreign HOST. However, the problem is not any particular interaction scheme; the problem is that we expect many of these kinds of schemes to occur. We have not found any general solutions to these problems, but some observations and conjectures may lead the way.

With respect to heterogeneous consoles, we note that although consoles are rarely compatible, many are equivalent. It is probably reasonable to treat a model 37 teletype as the equivalent of an IBM 2741. Similarly, most storage scopes will form an equivalence class, and most refresh display scopes will form another. Furthermore, a hierarchy might emerge with members of one class usable in place of those in another, but not vice versa. We can imagine that any scope might be an adequate substitute for a teletype, but hardly the reverse. This observation leads us to wonder if a network-wide language for consoles might be possible. Such a language would provide for distinct treatment of different classes of consoles, with semantics ap-

propriate to each class. Each site could then write interface programs for its consoles to make them look like network standard devices.

Another observation is that a user evaluates an interactive system by comparing the speed of the system's responses with his own expectations. Sometimes a user feels that he has made only a minor request, so the response should be immediate; at other times he feels he has made a substantial request, and is therefore willing to wait for the response. Some interactive subsystems are especially pleasant to use because a great deal of work has gone into tailoring the responses to the user's expectations In the network, however, a local user level process intervenes between a local console and a foreign subsystem, and we may expect the response time for minor requests to degrade. Now it may happen that all of this tailoring of the interaction is fairly independent of the portion of the subsystem which does the heavy computing or I/O. In such a case, it may be possible to separate a subsystem into two sections. One section would be the "substantive" portion; the other would be a "front end" which formats output to the user, accepts his inputs, and controls computationally simple responses such as echoes. In the example above, the program to accumulate a line and generate echoes would be the front end of some subsystem. We now take notice of the fact that the local HOSTs have substantial computational power, but our current designs make use of the local HOST only as a data concentrator. This is somewhat ironic, for the local HOST is not only poorly utilized as a data concentrator, it also degrades performance because of the delays it introduces.

These arguments have led us to consider the possibility of a Network Interface Language (NIL) which would be a network-wide language for writing the front end of interactive subsystems. This language would have the feature that subprograms communicate through network-like connections. The strategy is then to transport the source code for the front end of a subsystem to the local HOST, where it would be compiled and executed.

During preliminary discussions we have agreed that NIL should have at least the following semantic properties not generally found in languages.

1. Concurrency. Because messages arrive asynchronously on different connections, and because user input is not synchronized with subsystem output, NIL must include semantics to accurately model the possible concurrencies.
2. Program Concatenation. It is very useful to be able to insert a program in between two other programs. To achieve this, the interconnection of programs would be specified at run time and would not be implicit in the source code.
3. Device substitutability. It is usual to define languages so that one device may be substituted for another. The requirement here is that any device can be modeled by a NIL program. For example, if a network standard display controller manipulates tree-structures according to messages sent to it then these structures must be easily implementable in NIL.

NIL has not been fully specified, and reservations have been expressed about its usefulness. These reservations hinge upon our conjecture that it is possible to divide an interactive subsystem into a transportable front end which satisfies a user's expectations at low cost and a more substantial stay-at-home section. If our conjecture is false, then NIL will not be useful; otherwise it seems worth pursuing. Testing of this conjecture and further development of NIL will take priority after low level HOST–HOST protocol has stabilized.

## HOST/IMP INTERFACING

The hardware and software interfaces between HOST and IMP is an area of particular concern to the HOST organizations. Considering the diversity of HOST computers to which a standard IMP must connect, the hardware interface was made bit serial and full-duplex. Each HOST organization implements its half of this very simple interface.

The software interface is equally simple and consists of messages passed back and forth between the IMP and HOST programs. Special error and signal messages are defined as well as messages containing normal data. Messages waiting in queues in either machine are sent at the pleasure of the machine in which they reside with no concern for the needs of the other computer.

The effect of the present software interface is the needless rebuffering of all messages in the HOST in addition to the buffering in the IMP. The messages have no particular order other than arrival times at the IMP. The Network Control Program at one HOST (e.g., Utah) needs waiting RFNM's before all other messages. At another site (e.g., SRI), the NCP could benefit by receiving messages for the user who is next to be run.

What is needed is coding representing the specific needs of the HOST on both sides of the interface to make intelligent decisions about what to transmit next over the channel. With the present software interface, the channel in one direction once committed to a particular message is then locked up for up to 80 milli-