Network Working Group
Request for Comments: 61

Dave Walden
Bolt Beranek and Newman

July 17, 1970

A Note on Interprocess Communication in a Resource Sharing Computer Network

The attached note is a draft of a study I am still working on. It may be of general interest to network participants.

Interprocess Communication in a Resource Sharing Computer Network

INTRODUCTION

"A resource sharing computer network is defined to be a set of autonomous, independent computer systems, interconnected so as to permit each computer system to utilize all of the resources of each other computer system. That is, a program running in one computer system should be able to call on the resources of the other computer systems much as it would normally call a subroutine." This definition of a network and the desirability of such a network is expounded upon by Roberts and Wessler in [1].

The actual act of resource sharing can be performed in two ways: in a pairwise ad hoc manner between all pairs of computer systems in the network or according to a systematic network wide standard. This paper develops one possible network wide system for resource sharing.

I believe it is natural to think of resources as being associated with processes [2] and therefore view the fundamental problem of resource sharing to be the problem of interprocess communication. I also share with Carr, Crocker, and Cerf [3] the view that interprocess communication over a network is a subcase of general interprocess communication in a multiprogrammed environment.

These views pervade this study and have led to a two part study. First, a model for a time-sharing system having capabilities particularly suitable for enabling interprocess communication is constructed. Next, it is shown that these capabilities can be easily used in a generalized manner which permits interprocess communication between processes distributed over a computer network.

This note contains ideas based on many sources. Particularly influential were -- 1) an early sketch of a Host protocol for the ARPA Network [1][3][4] by W. Crowther of Bolt Beranek and Newman Inc. (BBN) and S. Crocker of UCLA; 2) Ackerman and Plummer’s paper on the MIT PDP-1 time sharing system [5]; and 3) discussion with R. Kahn of BBN about Host protocol, message control, and routing for the ARPA Network. Hopefully, there are also some original ideas in this note. I alone am responsible for the collection of all of these ideas into the system described herein, and I am therefore responsible for any inconsistencies or bugs in this system.

It must be emphasized that this note does not represent an official BBN position on Host protocol for the ARPA Computer Network.

A MODEL FOR A TIME-SHARING SYSTEM

This section describes a model time-sharing system which I think is particularly suitable for performing interprocess communication. The basic structure of this model time-sharing system is not original [5][9].

The model time-sharing system has two pieces: the monitor and the processes. The monitor performs several functions, including switching control from process to process as appropriate (e.g., when a process has used "enough" time or when an interrupt occurs), managing core and the swapping medium, controlling the passing of control from one process to another (i.e., protection mechanisms), creating processes, caring for sleeping processes, etc.

The processes perform most of the functions normally thought of as being supervisor functions in a time-sharing system (system processes) as well as the normal user functions (user processes). A typical system process is the disc handler or the file system. For efficiency reasons it may be useful to think of system processes as being locked in core.

A process can call on the monitor to perform several functions: start another, equal, autonomous process (i.e., load a program or find a copy of a program somewhere that can be shared, start it, and pass it some initial parameters); halt the running process; put the current process to sleep pending a specified event; send a message to a specified process; become available to receive a message from a specified process; become available to receive a message from any process; send a message to a process able to receive from any process; and request a unique number. There undoubtedly should also be other monitor functions. It is left as an exercise to the reader to convince himself that the monitor he is saddled with can be made to provide these functions -- most can.

I will not concern myself with protection considerations here, but instead will assume all of the processes are "good" processes which never make any mistakes. If the reader needs a protection structure to keep in mind while he reads this note, the _capability_ system described in [5][6][7][8] should be satisfying.

We now look a little closer at the eight operations listed above that a process can ask the monitor to perform.

START. This operation starts another process.   It has two parameters -- some kind of identification for the program that is to be loaded and a parameter list for that program.   Once the program is loaded, it is started at its given entry point and passed its parameter list in some well known manner. The process will continue to exist until it halts itself.

HALT. This operation puts the currently running process to sleep pending the completion of some event. The operation has one parameter, the event to be waited for. Sample events are arrival of a hardware interrupt, arrival of a message from another process, etc. The process is restarted at the instruction after the SLEEP command. The monitor never unilaterally puts a process to sleep except when the process overflows its quantum.

RECEIVE. This operation allows another process to send a message to this process. The operation has four parameters: the port (defined below) awaiting the message, the port a message will be accepted from, a specification of the buffer available to receive the message, and a location of transfer to when the transmission is complete. [In other words, an interrupt location. Any message port may be used to allow interrupts, event channels, etc. The user programs what he wants.]

SEND. This operation sends a message to some other process. [I suppose a process could also send a message to itself.] It has four parameters: a port to send the message to, the port the message is being sent from, the message, and a location to transfer to when the transmission is complete.

RECEIVE ANY. This operation allows any process to send a message to this process. The operation has four parameters: the port awaiting the message, the buffer available to receive the message, a location to transfer to when the message is received, and a location where the port which sent the message may be noted.

SEND FROM ANY. This operation allows a process to send a message to a process able to receive a message from any process. It has the same four parameters as SEND. The necessity for this operation will be discussed below.

UNIQUE. This operation obtains a unique number from the monitor.

A _port_ is a particular data path to or from a process. All ports have an associated unique number which is used to identify the port. Ports are used in transmitting messages from one process to another in the following fashion. Consider two processes, A and B, wishing to communicate. Process A executes a RECEIVE at port N from port M.

Process B executes a SEND to port N from port M. The monitor matches up the port numbers and transfers the message from process B to process A. As soon as the buffer has been fully transmitted out of process B, process B is restarted at the location specified in the SEND operation. As soon as the message is fully received at process A, process A is restarted at the location specified in the RECEIVE operation. Just how the processes come by the correct port numbers with which to communicate with other processes is not the concern of the monitor -- this problem is left to the processes.

An example. Suppose that our model time-sharing system is initialized to have several processes always running. Additionally, these permanent processes have some universally known and permanently assigned ports. [Or perhaps there is only one permanently known port which belongs to a directory-process which keeps a table of permanent-process/well-known-port associations.] Suppose that two of the permanently running processes are the logger-process and the teletype-scanner-process. When the teletype-scanner-process first starts running, it puts itself to sleep awaiting an interrupt from the hardware teletype scanner. The logger-process initially puts itself to sleep awaiting a message from the teletype-scanner-process via well-known permanent SEND and RECEIVE ports. The teletype-scanner-process keeps a table indexed by teletype number containing in each entry a port to send characters from that teletype to, and a port at which to receive characters for that teletype. If a character arrives (waking up the teletype-scanner-process) and the process does not have any entry for that teletype, it gets a pair of unique numbers from the monitor (via UNIQUE) and sends a message containing this pair of numbers to the logger-process using the ports that the logger-process is known to have a RECEIVE pending for. [Actually, the scanner process could always use the same pair of port numbers for a particular teletype as long as they were passed on to only one copy of the executive at a time.] The scanner-process also enters the pair of numbers in the teletype table, and sends the characters and all future characters from this teletype to the port with the first number from the port with the second number. The scanner-process probably also passes a second pair of unique numbers to the logger-process for it to use for teletype output and does a RECEIVE using these numbers. The logger-process when it receives the message from the scanner-process, starts up a copy of what SDS 940 TSS [12] users call the executive (that program which prints file directories, tells who is on other teletypes, runs subsystems, etc.) and passes this copy of the executive, the port numbers so this executive-process can also do its in’s and out’s to the teletype using these ports. If the logger-process wants to get a job number and password from the user, it can temporarily use the port numbers to communicate with the user before it passes them on to the executive.

Port numbers are often passed among processes. More rarely, a port is transferred to another process. It is crucial that once a process transfers a _port_ to some other process that the first process no longer use the port. We could add a mechanism that enforces this. The protected object system of [8] is one such mechanism. [Of course, if the protected object system is available to us, there is really no need for two port numbers to be specified before a transmission can take place. The fact that a process knows an existing RECEIVE port number is prima facie evidence of the process' right to send to that port. The difference between RECEIVE and RECEIVE ANY ports then depends solely on the number of copies of a particular port number that have been passed out. A system based on this approach would clearly be preferable to the one described here if it was possible to assume all of the autonomous time-sharing system in a network would adopt this protection mechanism. If this assumption cannot be made, it seems more practical to require both port numbers.]

Note that somewhere in the monitor there must be a table of port numbers associated with processes and restart locations. The table entries are cleared after each SEND/RECEIVE match is made. Also note that if a process is running (perhaps asleep), and has RECEIVE ANY pending, then any process knowing the receive port number can talk to that process without going through loggers or any of that. This is obviously essential within a local time-sharing system and seems very useful in a more general network if the ideal of resource sharing is to be reached.

When a SEND is executed, nothing happens until a matching RECEIVE is executed. If a proper RECEIVE is not executed for some time the SEND is timed out after a while and the SENDing process is notified. If a RECEIVE is executed but the matching SEND does not happen for a long time, the RECEIVE is timed out and the RECEIVing process is notified.

A RECEIVE ANY never times out, but may be taken back. A SEND FROM ANY message is always sent immediately and will be discarded if a proper receiver does not exist. An error message is not returned and acknowledgment, if any, is up to the processes. If the table where the SEND and RECEIVE are matched up ever overflows, a process originating a further SEND or RECEIVE is notified just as if the SEND or RECEIVE timed out.

Generally, well known, permanently assigned ports are used via RECEIVE ANY and SEND FROM ANY. The permanent ports will most often be used for starting processes going and consequently little data will be sent via them.

Still another example, this time a demonstration of the use of the FORTRAN compiler. We have already explained how a user sits down at his teletype and gets connected to an executive. We go on from there. The user is typing in and out of the executive which is doing SENDs and RECEIVEs. Eventually the user types RUN FORTRAN and the executive asks the monitor to start up a copy of the FORTRAN compiler and passes to FORTRAN as start up parameters the two ports the executive was using to talk to the teletype. FORTRAN is of course expecting these parameters and does SENDs and RECEIVEs to these ports to discover what input and output files the user wants to use. FORTRAN types INPUT FILE? to the user who responds F001. FORTRAN then sends a message to the file-system-process which is asleep waiting for something to do. The message is sent via well-known ports and it asks the file system to open F001 for input. The message also contains a pair of ports that the file-system-process can use to send its reply. The file-system looks up F001, opens it for input, makes some entries in its open file tables, and sends a message back to FORTRAN which contains the ports which FORTRAN can use to read the file. The same procedure is followed for the output file. When the compilation is complete, FORTRAN returns the teletype port numbers back to the executive which has been asleep waiting for a message from FORTRAN, and then FORTRAN halts itself. The file-system-process goes back to sleep when it has nothing else to do.

[The reader should have noticed by now that I do not like to think of a new process (consisting of a new conceptual copy of a program) being started up each time another user wishes to use the program. Rather, I like to think of the program as a single process which knows it is being used simultaneously by many other processes and consciously multiplexes among the users or delays service to users until it can get around to them.]

Again, the file-system-process can keep a small collection of port numbers which it uses over and over if it can get file system users to return the port numbers when they are done with them. Of course, when this collection of port numbers has eventually dribbled away, the file system can get some new unique numbers from the monitor.

Note that when two processes wish to communicate they set up the connection themselves, and they are free to do it in a mutually convenient manner. For instance, they can exchange port numbers or one process can pick all the port numbers and instruct the other process which to use. Of course, in a particular implementation of a time-sharing system, the builders of the system might choose to restrict the processes’ execution of SENDs and RECEIVEs and might forbid arbitrary passing around of port numbers, requiring instead that the monitor be called (or some other special program) to perform these functions.

Flow control is provided in this system by the simple method of never starting a SEND from one process until a RECEIVE is executed by the receiver. Of course, interprocess messages may be sent back and forth suggesting that a process stop sending or that space be allocated, etc.

INTERPROCESS COMMUNICATION BETWEEN REMOTE PROCESS

The system described in the previous section easily generalizes to allow interprocess communication between processes at geographically different locations as, for example, within a computer network.

Consider first a simple configuration of processes distributed around the points of a star. At each point of the star there is an autonomous time-sharing system. A rather large, smart computer system, called the Network Controller, exists at the center of the star. No processes can run in this center system, but rather it should be thought of as an extension of the monitor of each time-sharing system in the network.

It should be obvious to the reader that if the Network Controller is able to perform the operations SEND, RECEIVE, SEND FROM ANY, RECEIVE ANY, and UNIQUE and that if all of the monitors in all of the time-sharing systems in the network do not perform these operations themselves but rather ask the Network Controller to perform these operations for them, then we have solved the problem of interprocess communication between remote processes. We have no further change to make.

The reason everything continues to work when we postulate the existence of the Network Controller is that the Network Controller can keep track of which RECEIVEs have been executed and which SENDs have been executed and match them up just as the monitor did in the model time-sharing system. A networkwide port numbering scheme is also possible with the Network Controller knowing where (i.e., at which site) a particular port is at a particular time.

Next, consider a more complex network in which there is no common center point making it necessary to distribute the functions performed by the Network Controller among the network nodes. In the rest of this section I will show that it is possible to efficiently and conveniently distribute the functions performed by the star Network Controller among the many network sites and still enable general interprocess communication between remote processes.

Some changes must be made to each of the four SEND/RECEIVE operations described above to adapt them for use in a distributed network. To RECEIVE is added a parameter specifying a site to which the RECEIVE is to be sent. To SEND FROM ANY and SEND is added a site to send the SEND to although this is normally the local site. Both RECEIVE and RECEIVE ANY have added the provision for obtain the source site of any received message. Thus, when a RECEIVE is executed, the RECEIVE is sent to the site specified, possibly a remote site. Concurrently a SEND is sent to the same site, normally the local site of the process executing the SEND. At this site, called the rendezvous site, the RECEIVE is matched with the proper SEND and the message transmission is allowed to take place to the site from whence the RECEIVE came.

A RECEIVE ANY never leaves its originating site and therein lies the necessity for SEND FROM ANY. It must be possible to send a message to a RECEIVE ANY port and not have the message blocked waiting for RECEIVE at the sending site. Of course, it would be possible to construct the system so the SEND/RECEIVE rendezvous takes place at the RECEIVE site and eliminate the SEND FROM ANY operation, but in my judgment the ability to block a normal SEND transmission at the source site more than makes up for the added complexity.

Somewhere at each site a rendezvous table is kept. This table contains an entry for each unmatched SEND or RECEIVE received at that site and also an entry for all RECEIVE ANYs given at that site. A matching SEND/RECEIVE pair is cleared from the table as soon as the match takes place or perhaps when the transmission is complete. As in the similar table kept in the model time-sharing system, SEND and RECEIVE entries are timed out if unmatched for too long and the originator is notified. RECEIVE ANY entries are cleared from the table when a fulfilling message arrives.

The final change necessary to distribute the Network Controller functions is to give each site a portion of the unique numbers to distribute via its UNIQUE operation. I’ll discuss this topic further below.

To make it clear to the reader how the distributed Network Controller works, an example follows. The details of what process picks port numbers, etc. are only exemplary and are not a standard specified as part of the system.

Suppose there are two sites in the network: K and L. Process A at site K wishes to communicate with process B at site L. Process B has a RECEIVE ANY pending at port M.

```
                 SITE K              SITE L
               ________            ________
              /        \          /        \
             /          \        /          \
            /            \      /            \
           |  Process A   |    |   Process B  |
           |              |    |              |
           |              |    |              |
            \            /      \            /
             \          /        \   port M /
              \________/          \____^___/
                                       |
                                   RECEIVE ANY
```

Process A, fortunately, knows of the existence of port M at site L
and sends messages using the SEND FROM ANY operation from port N to
port M. The message contains two port numbers and instructions for
process B to SEND messages to process A to port P from port Q. Site
K’s site number is appended to this message along with the message’s
SEND port N.

```
                SITE K                        SITE L
              ________                      ________
             /        \                    /        \
            /          \                  /          \
           /            \                /            \
          |  Process A   |              |   Process B  |
          |              |              |              |
          |              |              |              |
           \            /                \            /
            \  port N  /--->SEND FROM --->\  port M  /
             \________/       ANY          \________/

                        to port M, site L
                        containing K, N, P, & Q
```

Process A now executes a RECEIVE at port P from port Q. Process A
specifies the rendezvous site to be site L.

```
                    SITE K                         SITE L
                  ________                 R    ________
                 /        \                e   /        \
                /          \               n T/          \
               /            \              d a            \
              |              |             e b  Process B |
              |  Process A   |             z l            |
              |              |             v e            |
               \            /              o \            /
                \  port P  /  RECEIVE ---> u  \          /
                 \________/   MESSAGE      s   \________/

                                to site L
                                containing P, Q, & K
```

A RECEIVE message is sent from site K to site L and is entered in the rendezvous table at site L. At some other time, process B executes a SEND to port P from port Q specifying site L as the rendezvous site.

```
                    SITE K                         SITE L
                  ________                 R    ________
                 /        \                e   /        \
                /          \               n T/          \
               /            \              d a            \
              |              |             e b  Process B |
              |  Process A   |             z l            |
              |              |             v e            |
               \            /              o \            /
                \  port P  /               u <--- port Q /
                 \________/    SEND        s   \________/
                                to site L
                                containing P & Q
```

A rendezvous is made, the rendezvous table entry is cleared, and the transmission to port P at site K takes place. The SEND site number (and conceivably the SEND port number) are appended to the messages of the transmission for the edification of the receiving process.

```
                SITE K                                SITE L
              ________                              ________
             /        \                            /        \
            /          \                          /          \
           /            \                        /            \
          |   Process A  |                      |  Process B   |
          |              |                      |              |
          |              |                      |              |
           \  port P    /                        \  port Q    /
            \          / <---- transmission <---- \          /
             \________/    to port T, site K       \________/
                            containing data and L
```

Process B may simultaneously wish to execute a RECEIVE from port N at port M.

Note that there is only one important control message in this system which moves between sites, the type of message that is called a Host/Host protocol message in [3]. This control message is the RECEIVE message. There are two other possible intersite control messages: an error message to the originating site when a RECEIVE or SEND is timed out, and the SEND message in the rare case when the rendezvous site is not the SEND site.

Of course there must also be a standard format for messages between ports. For example, the following:

```
    +-----------------+  +-----------------+  +-----------------+
    | rendezvous site |  | destination site|  |  source site    |
    +-----------------+  +-----------------+  +-----------------+
    | RECEIVE port    |  | RECEIVE port    |  |  RECEIVE port   |
    +-----------------+  +-----------------+  +-----------------+
    | SEND port       |  | SEND port       |  |   SEND port     |
    +-----------------+  +-----------------+  +-----------------+
    |                 |  | source port     |  |                 |
    |                 |  +-----------------+  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |     data        |  |      data       |  |     data        |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    |                 |  |                 |  |                 |
    +-----------------+  +-----------------+  +-----------------+
       transmitted          transmitted          received
       by SEND              by Network           by RECEIVE
       process              Controller           process
```

Note: for a SEND FROM ANY message, the rendezvous site is the destination site.

In the model time-sharing system it was possible to pass a port from process to process. This is still possible with a distributed Network Controller. [The reader unconvinced of the utility of port passing is directed to read the section on reconnection in [11].]

Remember that for a message to be sent from one process to another, a SEND to port M from port N and a RECEIVE at port M from port N must rendezvous, normally at the SEND site. Both processes keep track of where they think the rendezvous site is and supply this site as a parameter of appropriate operations. The RECEIVE process thinks it is the SEND site and the SEND process normally thinks it is the SEND site also. Since once a SEND and a RECEIVE rendezvous, the transmission is sent to the source of the RECEIVE and the entry in the rendezvous table is cleared and must be set up again for each further transmission from N to M, it is easy for a RECEIVE port to be moved. If a process sends both the port numbers and the rendezvous site number to a new process at some other site which executes a RECEIVE using these same old port numbers and rendezvous site specification, the SENDer never knows the RECEIVEr has moved. It is slightly harder for a SEND port to move. However, if it does, the pair of port numbers that has been being used for a SEND and the original rendezvous site number are passed to the new site. The process at the new SEND site specifies the old rendezvous site with the first SEND from the new site. The RECEIVE process will also still think the rendezvous site is the old site, so the SEND and RECEIVE will meet at the old site. When they meet, the entry in the table at that site is cleared, the rendezvous site number for the SEND message is changed to the site which originated the SEND message and both the SEND and RECEIVE messages are sent to the new SEND site just as if they had been destined for there in the first place. The SEND and RECEIVE then meet again at the new rendezvous site and transmission may continue as if the port had never moved. Since all transmissions contain the source site number, further RECEIVEs will be sent to the new rendezvous site. It is possible to discover that this special manipulation must take place because a SEND message is received at a site which did not originate the SEND message. Everything is so easily changed because there are no permanent connections to break and move as in the once proposed reconnection scheme for the ARPA network [10][11] that is, connections only exist fleetingly in the system described here and can therefore be remade between any pair of processes which at any time happen to know each other's port numbers and have some clue where they each are.

Of course, all of this could have been done by the processes sending messages back and forth announcing any potential moves and the new site numbers.

REFERENCES

[1] L. Roberts and B. Wessler, Computer Network Development to achieve Resource Sharing, Proceedings 1970 SJCC.

[2] V. Vyssotsky, F. F. Corbato, and R. Graham, Structure of the MULTICS Supervisor, Proceedings 1965 FJCC.

[3] C. Carr, S. Crocker, and V. Cerf, Host/Host Communication Protocol in the ARPA Network, Proceedings 1970 SJCC.

[4] F. Heart, et al, The Interface Message Processor for the ARPA Computer Network, Proceedings 1970 SJCC.

[5] W. Ackerman and W. Plummer, An Implementation of Multi-processing Computer System, Proceedings Gatlinburg Symposium on Operating System Principles.

[6] J. Dennis and E. Van Horn, Programming Semantics for Multiprogramming Computation, Proceedings of the San Dimes Conference on Programming Language and Pragmatics.

[7] B. Lampson, Dynamic Protection Structures, Proceedings FJCC 1969.

[8] B. Lampson, An Overview of the CAL Time-Sharing System, Computer Center, University of Calif., Berkeley.

[9] P. Hansen, The Nucleus of a Multiprogramming System, CACM, April 1970.

[10] S. Crocker, ARPA Network Working Group Note #36.

[11] J. Postel and S. Crocker, ARPA Network Working Group Note #48.

[12] B. Lampson, 940 Lectures.

APPENDIX: AN APPLICATION

Only one resource sharing computer network currently exists, the aforementioned ARPA network. In this Appendix, I hope to show that the system that was described in this note can be applied to the ARPA network. A significant body of work exists on interprocess communication within the ARPA network. This work comes in several almost distinct pieces: the Host/IMP protocol, IMP/IMP protocol, and the Host/Host protocol. I assume familiarity with this work in the subsequent discussion. [See references [1][3][4][10][11]; Specifications for the Inter-connection of a Host to an IMP, BBN Report No. 1822; and ARPA Network Working Group Notes #37, 38, 39, 42, 44, 46, 47, 48, 49, 50, 54, 55, 56, 57, 56, 59.]

In the ARPA network, the IMP's have sole responsibility for correctly transmitting bits from one site to another. The Hosts have sole responsibility for making interprocess connections. Both the Host and IMP are concerned and take a little responsibility for flow control and message sequencing. Application of the interprocess communication system I have described leads me to different allocation of responsibility. The IMP still continues to correctly move bits from one site to another, but the Network Controller also resides in the IMP, and flow control is completely in the hands of the processes running in the Hosts although perhaps they use mechanisms provided by the IMPs.

The IMPs provide the SEND, RECEIVE, SEND FROM ANY, RECEIVE ANY, and UNIQUE operations in slightly altered forms for the Hosts and also maintain the rendezvous tables including moving of SEND ports when necessary.

It is perhaps easiest to step through the five operations again.

SEND. The Host gives the IMP a SEND port number, a RECEIVE port number, the rendezvous site, and a buffer specification=20 (e.g., start and end, beginning and length). The SEND is sent to the rendezvous site, normally the local site. When the matching RECEIVE arrives, the Host is notified of the RECEIVE port of the just arrived receive message. This port number is sufficient to identify the SENDing process although a given time-sharing system may have to keep internal tables mapping this port number into useful internal process identifiers. Simultaneously, the IMP will begin to ask the Host for specific chunks of the data buffer. These chunks will be sent off to the destination as the IMP's RFNM control allows. If a RFNM is not received for too long, implying a message has been lost in the network, the Host is asked for the same chunk of data again [which also allows messages to be completely thrown away by the IMP network if that should ever be useful], but the Host has the option to abort the transmission at this time. While a transmission is taking place, the Host may ask the IMP to perform other operations including other SENDs. A second SEND over a pair of ports already in the act of transmission is noted and the SEND becomes active as soon as the first transmission is complete. A third identical SEND results in an error message to the Host. If a SEND times out, an error is returned also.

RECEIVE. The Host gives the IMP a SEND port, a RECEIVE port, a rendezvous site, and a buffer description. The RECEIVE message is sent to the rendezvous site. When chunks of a transmission arrive for the RECEIVE port they are passed to the Host along with RECEIVE port number (and perhaps the SEND port number), and an indication to the Host where to put the data in its input buffer. When the last of the SEND buffer is passed into the Host, it is marked accordingly and the Host can then detect this. A second RECEIVE over the same port pair is allowed. A third results in an error message to the Host. The mechanism described in this and the previous paragraphs allows a pair of processes to always have both a transmission in progress and the next one pending. Therefore, no efficiency is lost. On the other hand, each transmission must be preceded by a RECEIVE into a specified buffer, thus providing complete flow control. (It is conceivable that the RECEIVE message could allocate a piece of network bandwidth while making its network traverse to the rendezvous site.)

RECEIVE ANY. The Host gives the IMP a RECEIVE port and a buffer descriptor. This works the same as RECEIVE but assumes the local site to be the rendezvous site.

SEND FROM ANY. The Host gives the IMP RECEIVE and SEND ports, the destination site, and a buffer descriptor. The IMP requests and transmits the buffer as fast as possible. A SEND FROM ANY for a non-existent port is discarded at the destination site.

RFNM’s are tied to the transmission of a particular chunk of buffer just as acknowledgments are now tied to packets and they perform the same function. If the Hosts allow the IMPs to reassemble buffers in the Hosts by the IMP telling the Host where it should put a buffer chunk as described above, chunks of a single buffer can be transmitted in parallel and several RFNMs can be outstanding simultaneously. Packet reassembly is still done in the IMPs.

A final operation must be provided by the IMP -- the UNIQUE operation. There are many ways to maintain unique numbers and three are presented here. The first possibility is for the Hosts to ask the IMPs for the unique numbers originally and then guarantee the integrity of any unique numbers currently owned by local processes and programs using whatever means the Host has at its disposal. In this case the IMPs would provide a method for a unique number to be sent from one host to another and would vouch for the number’s identity at the new site.

The second method is to simply give the unique numbers to the processes that are using them, depending on the non-malicious behavior of the processes to preserve the unique numbers, or if an accident should happen, the two passwords (SEND and RECEIVE ports) that are required to initiate a transmission. If the unique numbers are given out in a non-sequential manner and are reasonably long (say 32 bits) there is little danger.

In the final method, a user identification is included in the port numbers and the individual time-sharing systems guarantee the integrity of these identification bits. Thus a process, while not able to be sure that the correct port is transmitting to him, can be sure that some port of the correct user is transmitting. This is the so-called virtual net concept suggested by W. Crowther [3].

Random Contents. Putting these operations in the IMP requires the Host/Host protocol program to be written only once, rather than many times as is currently being done in the ARPA Network. The IMPs can stop a specific host transmission (by not asking for the next chunk for a while) if that should seem necessary to alleviate congestion problems in the communications subnet. And the IMP might know the approximate time it takes for a RECEIVE to get to a particular other site and warn the Host to wake up a process shortly before it becomes imminent that a message for that process will be arriving.