A System for Interprocess Communication in a Resource Sharing Computer Network

David C. Walden
Bolt Beranek and Newman Inc.*

A system of communication between processes in a time-sharing system is described and the communication system is extended so that it may be used between processes distributed throughout a computer network. The hypothetical application of the system to an existing network is discussed.

Key Words and Phrases: interprocess communication, time-sharing, computer networks, resource sharing
CR Categories: 3.81, 4.39, 4.82, 4.9

Copyright © 1972, Association for Computing Machinery, Inc. General permission to republish, but not for profit, all or part of this material is granted, provided that reference is made to this publication, to its date of issue, and to the fact that reprinting privileges were granted by permission of the Association for Computing Machinery.

* 50 Moulton Street, Cambridge, MA 02138. The work was primarily supported by the Advanced Research Projects Agency under Contract DAHC 15-69-C-0179.

1. Introduction

A resource sharing computer network is defined to be a set of autonomous, independent computer systems, interconnected to permit each computer system to utilize all the resources of the other computer systems much as it would normally call one of its own subroutines. This definition of a network and the desirability of such a network are expounded upon by Roberts and Wessler in [9]. Examples of resource sharing could include a program filing some data in the file system of another computer system, two programs in remote computer systems exchanging communications, or users simply utilizing programs of another computer system via their own.

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

It is useful to think of resources as being associated with processes[1] and available only through communication with these processes. This is a viewpoint which has been successfully applied to time-sharing systems [1] and has more recently been suggested to be an appropriate view for computer networks [8]. Consistent with this view, the fundamental problem of resource sharing is held in this paper to be the problem of inter-

[1] Almost any of the common definitions of a process would suit the needs of this paper.

process communication. With Carr, Crocker, and Cerf [2], the view is also held that interprocess communication over a network is a subcase of general interprocess communication in a multiprogrammed environment.

These views have led to a two-part study. First, a set of operations enabling interprocess communication within a single time-sharing system is constructed. This set of operations eschews many of the interprocess communication techniques currently in use within time-sharing systems—such as communication through shared memory—and relies instead on techniques that can be easily generalized to permit communication between remote processes. The second part of the study presents such a generalization. The hypothetical application of this generalized system to the ARPA Computer Network [9] is also discussed.

This paper does not represent a Bolt Beranek and Newman position on Host protocol for the ARPA network.

2. A System for Interprocess Communication Within a Time-Sharing System

This section describes a set of operations enabling interprocess communication within a time-sharing system. Following the notation of [10], this interprocess communication facility is called an IPC. As an aid to the presentation of this IPC, a model for a time-sharing system is described; the model is then used to illustrate the use of the interprocess communication operations.

The model time-sharing system has two pieces: the monitor and the processes. The monitor performs such functions as switching control from one process to another process when a process has used "enough" time, fielding hardware interrupts, 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, and providing to the processes a set of machine extending operations (often called "supervisor" or "monitor" calls).

The processes perform the normal user functions (user processes) as well as the functions usually thought of as being supervisor functions in a time-sharing system (system processes) but not performed by the monitor in the current model. A typical system process is the disk handler or the file system. System processes are probably allowed to execute in supervisor mode, and they actually execute I/O instructions and perform other privileged operations that user processes are not allowed to perform. In all other ways, user and system processes are identical. For reasons of efficiency, it may be useful to think of system processes as being locked in core.

Although they will be of concern later in this study, protection considerations are not our concern here; instead, all processes are assumed to be "good" processes which never make any mistakes. If the reader needs a protection structure to keep in mind while he reads this paper, the *capability* system developed in [1, 3, 6, and 7] should be satisfying.

Of the operations a process can call on the monitor to perform, the five following are of particular interest for providing a capability for interprocess communication.

RECEIVE. This operation allows a specified process to send a message to the process executing the RECEIVE. The operation has four parameters: the port (defined below) awaiting the message—the RECEIVE port; the port a message will be accepted from—the SEND port; a specification of the buffer available to receive the message; and a location to transfer to when the transmission is complete—the restart location.

SEND. This operation sends a message from the process executing the SEND to a specified process. It has four parameters: a port to send the message to—the RECEIVE port; the port the message is being sent from—the SEND port; a specification of the buffer containing the message to be sent; and the restart location.

RECEIVE ANY. This operation allows *any* process to send a message to the process executing the RECEIVE ANY. The operation has four parameters: the port awaiting the message—the RECEIVE port; a specification of the buffer available to receive the message; a restart location; and a location where the port which sent the message may be noted.

SLEEP. This operation allows the currently running process to put itself to sleep pending the completion of an event. The operation has one optional parameter—an event to be waited for. An example event is the arrival of a hardware interrupt. The monitor never unilaterally puts a process to sleep as a result of the process executing one of the above four operations; however, if a process is asleep when one of the above four operations is satisfied, the process is awakened.

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

A *port* is a particular data path to a process (a RECEIVE port) or from a process (a SEND port), and all ports have an associated unique *port number* which is used to identify the port. Ports are used in transmitting messages from one process to another in the following manner. Consider two processes, A and B, that wish to communicate. Process A executes a RECEIVE to 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.[2] How the processes come by the correct port numbers with which to communicate with other processes is discussed later.

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.* 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 using a supervisor call.

The mechanism of timing out "unused" table entries is of little fundamental importance—it merely provides a convenient method of garbage-collecting the table. There is no problem if an entry is timed out prematurely, because the process can always reexecute the operation. However, the timeout interval should be long enough so that continual reexecution of an operation will cause little overhead. 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.

The *restart location* is an interrupt entrance associated with a pseudo interrupt local to the process executing the operation specifying the restart location. If the process is running when the event causing the pseudo interrupt occurs (for example, a message arrives satisfying a pending RECEIVE), the effect is exactly as if the hardware interrupted the process and transferred control to the restart location. Enough information is saved for the process to continue execution at the point it was interrupted after the interrupt is serviced. If the process is asleep, it is readied and the pseudo interrupt is saved until the process runs again, and the interrupt is then allowed. Any RECEIVE or RECEIVE ANY message port may thus be used to provide process interrupts, event channels, process synchronization, message transfers, etc. The user programs what he wants.

Most time-sharing systems could be made to provide the five operations described above with relative ease, since these are only additional supervisor calls.

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.[3] 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 pair of port numbers to use to send characters from that teletype to a process and a pair of port numbers to use to receive characters for that teletype from a process. 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 for which the logger-process is known to have a RECEIVE pending. The scanner-process also enters the pair of numbers in the teletype table, and sends the character and all future characters from this teletype to the port with the first number from the port with the second number. The scanner-process must also pass a second pair of unique numbers to the logger-process for it to use for teletype output and do a RECEIVE using these port numbers. When the logger-process receives the message from the scanner-process, it may, for instance, start up a copy of what SDS 940 TSS [5] users call the executive[4] and pass the port numbers to this copy of the executive, so that this executive-process can also do its inputs and outputs 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. The scanner-process could always use the same port numbers for a particular teletype as long as the numbers were passed on to only one copy of the executive at a time.

It is important to distinguish between the act of passing a port from one process to another and the act of passing a port *number* from one process to another. In the previous example, where characters from a particular teletype are sent either to the logger-process or an executive-process by the teletype-scanner-process, the SEND port always remains in the teletype-scanner-process, while the RECEIVE port moves from the logger-process to the executive-process. On the other hand, the SEND port number is passed between the logger-process and the executive-process to enable the RECEIVE process to do a RECEIVE from the correct SEND port. It is crucial that, once a process transfers a *port* to some other process, the first process no longer uses the port. A mechanism that enforces this, such as the protected object system of [7], could be added. Using this mechanism, a process executing a SEND would need a capability for the SEND port, and only one capability for this SEND port would exist in the system at any given time. Likewise a process executing a RECEIVE would be required to have a capability for the RECEIVE port, and only one capability for this RECEIVE port would exist at a given time. Without such a protection mechanism, a port implicitly moves from one process to another by the processes merely using the port at disjoint times, even if the port's number is never explicitly passed.

[2] Interestingly, there seems no reason to prevent (and some reasons for allowing) a process concurrently having several outstanding RECEIVEs from different ports to a given port. For instance, process A might execute a RECEIVE to port N from port R concurrently with the above RECEIVE to port N from port M.

[3] Or perhaps there is only one permanently known port, which belongs to a directory-process that keeps a table of permanent-process/well-known-port associations.

[4] That program which prints file directories, tells who is on other teletypes, runs subsystems, etc.

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 could be considered *prima facie* evidence of the process's 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 that all autonomous time-sharing systems 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 in the interprocess communication system (IPC) being described here, 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, or they can just *know* the correct port numbers to use.[5] However, 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 ports and port numbers, requiring instead that the monitor be called (or some other special program) to perform these functions. Generally, well-known permanently-assigned ports are used via RECEIVE ANY. The permanent ports will most often be used for starting processes, and consequently, little data will be sent via them. If a process is running (perhaps asleep) and has a RECEIVE ANY pending, then any process knowing the RECEIVE port number can talk to that process without going through loggers. 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. For instance, in a resource sharing network, the programs in the subroutine libraries at all sites might have RECEIVE ANYS always pending over permanently assigned ports with well-known port numbers. Thus, to use a particular network resource such as a matrix inversion subroutine at a site with special matrix manipulation hardware, a process running anywhere in the network can send a message to the matrix inversion subroutine containing the matrix to be inverted and the port numbers to be used for returning the results.

Control of data flow is provided in this IPC by the simple method of never starting data transmission resultant from a SEND from one process until a RECEIVE is executed by the receiver. Of course, interprocess messages may also be sent back and forth suggesting that a process stop sending or that space be allocated.

An additional example demonstrates 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 port numbers the executive was using to talk to the teletype. (Thus, at least conceptually, FORTRAN is passed a port at which to RECEIVE characters from the teletype and a port from which to SEND characters to the teletype.) FORTRAN is, of course, expecting these parameters and does SENDs and RECEIVEs via the indicated ports to discover from the user 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 port numbers 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 back to FORTRAN a message containing the port numbers that 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 (and the ports) back to the executive that has been asleep waiting for a message from FORTRAN and then halts itself. With nothing else to do, the file-system-process goes back to sleep.

3. A System for Interprocess Communication Between Remote Processes

The IPC 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 operating system.[6] 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 of the operating systems in the network.

If the Network Controller is able to perform the operations SEND, RECEIVE, RECEIVE ANY, and UNIQUE, and if all of the monitors in all of the operating systems in the network do not perform these operations themselves but rather ask the Network Controller to perform

[5] Called ipc-setup in [10].

[6] "Operating system" rather than "time-sharing system" is used in this section to point up the fact that the autonomous systems at the network nodes may be either full-blown time-sharing systems in their own right, an individual process in a larger geographically distributed time-sharing system, or merely autonomous sites wishing to communicate.

these operations for them, then the problem of interprocess communication between remote processes is solved. No further changes are necessary since 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 network-wide 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 it will be shown that the set of functions performed by the star Network Controller can be replicated at each of the network sites so that efficient and convenient general interprocess communication between remote processes is still possible. We call all of these network controllers taken together a *distributed Network Controller*.

Some changes must be made to each of the three SEND/RECEIVE operations described above to adapt them for use in a distributed Network Controller. To RECEIVE is added a parameter specifying a site to which the RECEIVE is to be sent. To SEND messages is added a site to which the SEND may be sent, although this is normally the local site. Both RECEIVE and RECEIVE ANY have added the provision for obtaining 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. At some (other) time a SEND appears at the same site, normally the local site of the process executing the SEND. (That is, the RECEIVE goes to the SEND site; the SEND does not go anywhere.) At this site, called the *rendezvous site*, the RECEIVE is matched with the proper SEND, and the message transmission is allowed to take place from the SEND site to the site from where the RECEIVE came.

A RECEIVE ANY never leaves its originating site; however, since it must be possible to send a message to a RECEIVE ANY port and not have the message blocked waiting for a RECEIVE at the sending site, we must introduce a new operation which will be called FORCE SEND. The FORCE SEND operation has the same parameters as SEND (a port to send the message to—the RECEIVE port; the port the message is being sent from—the SEND port; a specification of the buffer containing the message to be sent; the restart location; and the site to which to send the message). The message resultant from a FORCE SEND is always sent immediately to the rendezvous site where it is discarded if a proper RECEIVE ANY is not found. An error message is not returned, and acknowledgment, if any, is up to the processes. It is possible to construct a system so that the SEND/RECEIVE rendezvous takes place at the RECEIVE site and eliminates the FORCE SEND operation, but the ability to block a normal SEND transmission at the source site is probably worth the added complexity.

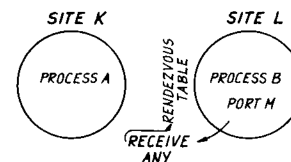
At each site a rendezvous table is kept. This table contains an entry for each unmatched SEND or RECEIVE received at that site and also an entry for all RECEIVE ANYs given at that site. A matching SEND/RECEIVE pair is cleared from the table as soon as the match takes place. As in the similar table kept 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 is to give each site a portion of the unique numbers to distribute via its UNIQUE operation. This topic will be discussed further below.

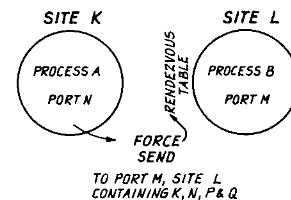
All interprocess communication at a site, to both local and remote processes, should be done via the local portion of the distributed Network Controller. The case is less clear in a star network with a central Network Controller where there is a trade-off between efficiency if local interprocess communication does not have to go through the central Network Controller and increased versatility if it does. The correct solution might be to distribute the Network Controller even for star networks.

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 IPC.

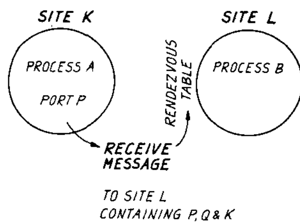
Suppose that, for two sites in the network, K and L, process A at site K wishes to communicate with process B at site L. Process B has a RECEIVE ANY pending at port M.



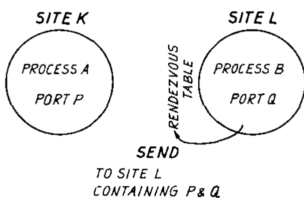
Process A, fortunately, knows of the existence of port M at site L and sends a message using the FORCE SEND operation from port N to port M. The message contains two port numbers and instructions for process B to SEND messages for process A to port P from port Q. Site K's site number is appended to this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



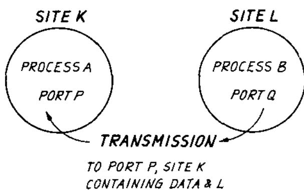
Process A now executes a RECEIVE at port P from port Q. Process A specifies the rendezvous site to be site L.



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.

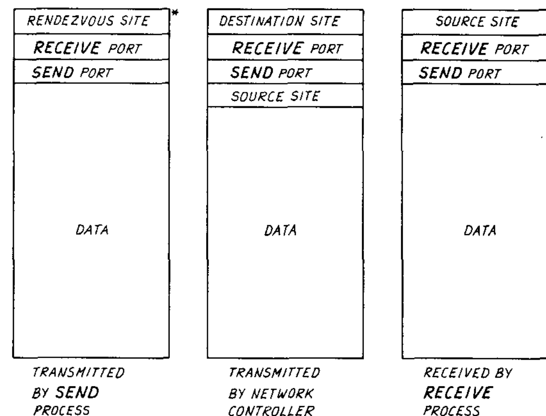


A rendezvous is made, the rendezvous table entry is cleared, and the transmission to port P at site K takes place. The SEND site number (and conceivably the SEND port number) is appended to the messages of the transmission for the edification of the receiving process.



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

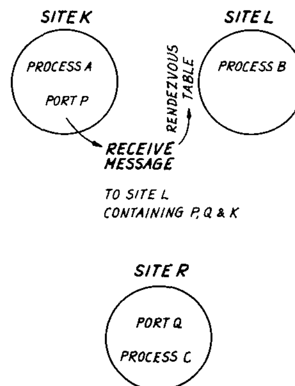
Note that there is only one important control message in this system which moves between sites—the type of message that is called a Host/Host protocol message in [2]. This control message is the RECEIVE message. There are two other possible intersite control messages: (1) a message to the originating site when a RECEIVE or SEND is timed out, and (2) the SEND message in the rare case which we will soon see when the rendezvous site is not the SEND site. There must also be a standard format for messages between ports, for example, the following:



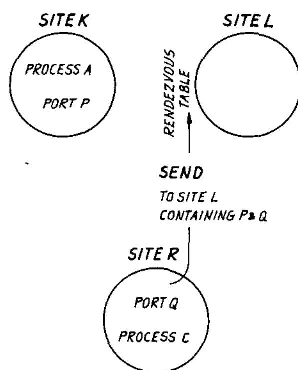
In the model time-sharing system it was possible to pass a port from process to process. This is still possible with a distributed Network Controller.

Remember, that for a message to be sent from one process to another, a SEND and a matching RECEIVE must rendezvous. 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, 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.

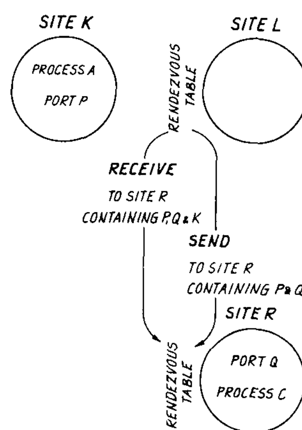
It is slightly harder for a SEND port to be moved. However, if it is, it can be done as follows: Suppose port Q from the preceding example moves from process B at site L to process C at site R. Process A will still think the rendezvous site is site L, and a RECEIVE message will still be sent there.



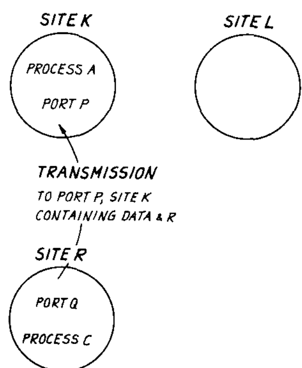
Process C specifies the old rendezvous site L, with the *first* SEND from the new site, R.



When the rendezvous is made at site L, the entry in the rendezvous table is cleared and both the SEND and RECEIVE messages are sent to the source of the SEND message, just as if they had been destined to go there all along.



After the SEND and RECEIVE meet again at the new rendezvous site, R, 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 that did not originate the SEND message. Note that the SEND port and the RECEIVE port can move concurrently.

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

A problem that may have occurred to the reader is how the SEND and RECEIVE buffers get matched for size. The easiest solution would be to require that all buffers have a common size, but this is unacceptable since it does not easily extend to a situation where processes in autonomous operating systems are attempting to communicate.

A second solution having great appeal due to its simplicity is for the processes to pass messages specifying buffer sizes. If this solution is adopted, excessive data sent from the SEND process and unable to fit into the RECEIVE buffer is discarded and the RECEIVE process notified.

A third solution would be for the RECEIVE buffer size to be passed to the SEND site with the RECEIVE message and to notify the SEND process when too much data is sent or even to pass the RECEIVE buffer size on to the SEND process. This last method would also permit the Network Controller at the SEND site to make two or more SENDs out of one, if that were necessary to match a small RECEIVE buffer size.

The maintenance of unique numbers is also a problem when the processes are geographically distributed. Of the three solutions to this problem presented here, the first is for the autonomous operating systems to ask the Network Controller for the unique numbers originally and then guarantee the integrity of any unique numbers currently owned by local processes and programs using whatever means are at the operating system's disposal. In this case, the Network Controller would provide a method for a unique number to be sent from one site to another and would vouch for the number's identity at the new site.

The second method is simply to give the unique numbers to the processes that are using them, depending on the nonmalicious behavior of the processes to preserve the unique numbers, or if an accident should happen, the *two* passwords (SEND and RECEIVE port numbers) that are required to initiate a transmission. If the unique numbers are given out in a nonsequential 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 operating 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.

A third difficult problem arises when remote processes wish to communicate—the problem of maintaining high bandwidth connections between the remote processes. The solution to this problem lies in allowing

the processes considerable information about the state of an ongoing transmission.

First, we examine a SEND process in detail. When a process executes a SEND, the local portion of the Network Controller passes the SEND on to the rendezvous site, normally the local site. When a RECEIVE arrives matching a pending SEND, the Network Controller notifies the SEND process by causing an interrupt to the specified restart location. Simultaneously, the Network Controller starts shipping the SEND buffer to the RECEIVE site. When transmission is complete, a flag is set which the SEND process can test. While a transmission is taking place, the process may ask the Network Controller 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 SENDing process.

Next, we examine a RECEIVE process in detail. When a process executes a RECEIVE, the RECEIVE is sent to the rendezvous site. When data resultant from this RECEIVE starts to arrive at the RECEIVE site, the RECEIVE process is notified via an interrupt to the specified restart location. When the transmission is complete, a flag is set which the RECEIVE process can test. A second RECEIVE over the same port pair is allowed. A third results in an error message to the RECEIVE process. Thus there is sufficient machinery to allow a pair of processes always to 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 continuing to provide control of data flow.

4. A Hypothetical Application

Only one resource sharing computer network currently exists, the ARPA Computer Network. This section is a discussion of the hypothetical application of the system described in this paper to the ARPA Network [2, 4, 9].

The ARPA Network currently incorporates over 20 sites spread across the United States. Each site consists of one to three (potentially four) independent computer systems called Hosts and one communications computer system called an IMP. All of the Hosts at a site are directly connected to the IMP. The IMPS themselves are connected together by 50-kilobit phone lines (much higher rate lines are a potential), although each IMP is connected to only one to five other IMPS. The IMPS provide a communications subnet through which the Hosts communicate. Data is sent through the communications subnet in messages less than (about) 8,100 bits long. When a message is received by the IMP at the destination site, that IMP sends an acknowledgment, called a RFNM, (Request for Next Message), to the source site. To allow more than one "logical transmission" to be in progress at a time between a particular pair of destination and source Hosts, each message is given a *link number*, and all messages in one logical transmission are sent with the same link number.

A system for interprocess communication for the ARPA Network (let us call this IPC for ARPA) has already been designed by the ARPA Network Working Group, under the chairmanship of S. Crocker of ARPA, and this design is currently being refined and implemented.[7] In contrast to the IPC of this paper, in IPC for ARPA, before two processes can communicate, a *connection* must be set up between a send socket in one process's monitor and a receive socket in the other process's monitor (a *socket* is an element in a network-wide name space into which each monitor maps its own internal port name space). For two processes to make connection, each process makes a connection request to its own monitor. The two monitors then exchange these requests via messages over the *control link*, a special link between each pair of Hosts which is always reserved for control messages. If both monitors are in agreement, the connection is established and a link is assigned for use by the new connection. This new connection exists until it is explicitly terminated, again using intermonitor control messages over the control link. During the "life" of the connection, many messages may be sent from the port at one end to the port at the other, and all these messages are sent over the link assigned to the connection. However, since the connection exists over a series of many messages, a mechanism is provided to stop the flow of messages when the receiving process's Host is overloaded. This mechanism is a system whereby the sender is notified of the receiver's buffering capacity as part of the connection setup, and the sender stops transmitting when it has transmitted enough messages to exhaust the capacity, unless a control message carrying notification of replenishment of the buffering capacity at the receiver arrives first.

The IPC for ARPA comes in several amost distinct pieces: The Host/IMP protocol, the IMP/IMP protocol, and the Host/Host protocol. The IMPS 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.

[7] This work is documented in a series of working group notes which, with the exception of [2], have unfortunately not found their way into the open literature.

[8] This also allows messages to be completely thrown away by the IMP subnet if that should ever be useful.

[9] This is in sharp contrast to IPC for ARPA which seems to utilize software "line switching" on top of a network designed in large part to demonstrate the utility of "message switching."

Application of the IPC described in this paper to the ARPA network suggests a different allocation of responsibility: The IMP still continues to move bits from one site to another correctly, but the Network Controller also resides in the IMP (whereas in the IPC for ARPA it resides in the Host), and flow control is completely in the hands of the processes running in the Hosts, although using the mechanisms provided by the Network Controllers in the IMPS.

The SEND, RECEIVE, FORCE SEND, RECEIVE ANY, and UNIQUE operations are implemented in the IMPS and are available to each IMP's Host. The IMPS also maintain the rendezvous tables, including moving of SEND ports when necessary. 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—a significant saving notwithstanding the still necessary Host/IMP software interface. It is the author's belief that the existing IMPS could probably be made to include the Network Controller with the addition of 4K of core.

It is perhaps useful 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 (e.g. start and end, or beginning and length). The SEND is sent to the rendezvous site IMP, normally the local IMP. When a matching RECEIVE arrives at the local IMP, the Host is notified of the RECEIVE port of the just-arrived message. This port number is sufficient to identify the SENDing process, although a given operating system may have to keep internal tables mapping this port number into a useful internal process identifier.

Simultaneously, the IMP begins to ask the Host for specific pieces of the SEND buffer and to send these pieces as network messages to the destination site. If a RFNM is not received for too long, implying a network message has been lost in the network, the Host is asked for the same data again and it is retransmitted.[8] Except for the last piece of a buffer, the IMP requests pieces from the Host which are common multiples of the word size of the source Host, IMP, and destination Host. This avoids mid-transmission word alignment problems.

RECEIVE. The Host gives the IMP a SEND port, a RECEIVE port, a rendezvous site, and a buffer descriptor. The RECEIVE message is sent to the rendezvous site. As the network messages making up a transmission arrive for the RECEIVE port, they are passed to the Host along with the RECEIVE port number (and perhaps the SEND port number), and (using the buffer descriptor) an indication to the Host where to put this data in its input buffer. When the last network message of the SEND buffer is passed into the Host, it is marked accordingly and the Host can then detect this. (It is conceivable that the RECEIVE message could also 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.

FORCE SEND. 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 FORCE SEND for a nonexistent port is discarded at the destination site.

In the ARPA Network, the Hosts are required by the IMPS to physically break their transmissions into messages, and successive messages of a single transmission must be delayed until the RFNM is received for the previous message. In the system described here, since RFNMS are tied to the transmission of a particular piece of buffer and since the Hosts allow the IMPS to reassemble buffers in the Hosts by the IMP telling the Host where to put each buffer piece, pieces of a single buffer can be transmitted in parallel network messages and several RFNMS can be outstanding simultaneously. This enables the Hosts to deal with transmissions of more natural sizes and higher bandwidth for a single transmission.

5. Conclusion

Since the system described in this paper has not been implemented, there are no clearly demonstrable conclusions and no performance reports that can be presented. Instead, we note what the author regards as the two significant features of the IPC system.

The first significant feature of this IPC is that connections are not maintained over a sequence of messages but instead are set up and expire on a message-by-message basis.[9] There are several advantages to this approach:

1. There is no need for a large number of intermonitor control messages to set up and break connections and consequently no need for a special "out of band" control channel.
2. The RECEIVing process has the opportunity after *each* message to stop the flow of messages. Further, since the RECEIVE process must provide a buffer, the monitor is relieved of the task of providing an indefinitely large buffering capacity.
3. Because connections exist only fleetingly, relatively complex operations such as moving a port are easy.
4. Errors have a minimal effect. For instance, if a

RECEIVE message is lost, the RECEIVing monitor will time it out and the process can do another RECEIVE just as is normally done when the SENDER refuses to SEND in response to a RECEIVE.
5. This IPC is suitable for implementation on small computers as well as large computers due to its simplicity and the nonnecessity for a large buffering capacity in the monitor.

There are also some disadvantages to this IPC; for instance:
1. There is a large amount of overhead associated with each message (e.g. port numbers).
2. To match the speed of a system based on prolonged connections, (e.g. IPC for ARPA) a significant amount of complexity had to be introduced.

The second significant feature of this IPC is the emphasis upon communication between processes and therefore the resources themselves, rather than between monitors. In the model time-sharing system described in Section 2 all possible functions (resources) were given to processes, and in Section 3 the processes (resources) control their own interprocess communication. While this seems to the author to be the proper emphasis if one's goal is flexible resource sharing, it does introduce new problems. For instance, many readers of early versions of this paper have asked, "How does a user who uses a remote process without going through the remote system's logger get billed?" There are ad hoc answers to this and probably to other infirmities in the IPC. (One answer to this particular question is that each process can send off a message to the accounting process whenever it finds itself being used.) However, we conclude with a more general *assertion* about the solutions to such questions.

Elegant solutions to many problems that will arise as computer networks are constructed and their effective utilization attempted will only be found if the computer networks are treated as single entities rather than as associations of autonomous systems. Far from allowing basically incompatible operating systems to communicate, an effective computer network will prove to require similar operating systems throughout the network, albeit running on machines of varying manufacture. General acceptance of this "fact" and its ramifications will bring closer the day when computer networks and resource sharing live up to their potential.

Acknowledgments. The author drew on many sources while developing the system suggested in this paper. Particularly influential were: (1) an early sketch of a Host protocol for the ARPA Network by S. Crocker of ARPA and W. Crowther of Bolt Beranek and Newman Inc. (BBN); (2) Ackerman and Plummer's paper [1] on the MIT PDP-1 time-sharing system; and (3) discussions with W. Crowther and R. Kahn of BBN about Host protocol, flow control, and message routing for the ARPA Network. B. Cosell of BBN and the referees made many incisive comments on the presentation of the paper.

Received August 1970; revised March 1971

References

1. Ackerman, W., and Plummer, W. An implementation of a multi-processing computer system. Proc. ACM Symp. on Operating System Principles, Gatlinsburg, Tenn., Oct. 1–4, 1967.
2. Carr, C., Crocker, S., and Cerf, V. Host/Host communication protocol in the ARPA network. Proc. AFIPS 1970 SJCC., Vol. 36, AFIPS Press, Montvale, N.J., pp. 589–597.
3. Dennis, J., and VanHorn, E. Programming semantics for multiprogrammed computations. *Comm. ACM 9*, 3 (Mar. 1966), 143–155.
4. Heart, F., Kahn, R., Ornstein, S., Crowther, W., and Walden, D. The interface message processor for the ARPA computer network. Proc., AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J. pp. 551–567.
5. Lampson, B. SDS 940 lectures, circulated informally.
6. Lampson, B. An overview of the CAL time-sharing system. Computer Center, U. of California, Berkeley.
7. Lampson, B. Dynamic protection structures. Proc. AFIPS 1969 FJCC, Vol. 35, AFIPS Press, Montvale, N.J., pp. 27–38.
8. Roberts, L. The ARPA Computer Network. Networks of Computers Symposium NOC-68. Proc. of Invitational Workshop, Ft. Meade, Md., National Security Agency, Sept. 1969.
9. Roberts, L., and Wessler, B. Computer network development to achieve resource sharing. Proc. AFIPS 1970 SJCC, Vol. 36, AFIPS Press, Montvale, N.J., pp. 543–549.
10. Spier, M., and Organick, E. The MULTICS interprocess communication facility. Proc. ACM Second Symp. on Operating Systems Principles, Princeton U., Oct. 20–22, 1969.