SOME CONSTRAINTS AND TRADEOFFS IN THE DESIGN OF NETWORK COMMUNICATIONS*

E. A. Akkoyunlu
K. Ekanadham
R. V. Huber†
Department of Computer Science
State University of New York at Stony Brook

A number of properties and features of interprocess communication systems are presented, with emphasis on those necessary or desirable in a network environment. The interactions between these features are examined, and the consequences of their inclusion in a system are explored. Of special interest are the time-out feature which forces all system table entries to "die of old age" after they have remained unused for some period of time, and the insertion property which states that it is always possible to design a process which may be invisibly inserted into the communication path between any two processes. Though not tied to any particular system, the discussion concentrates on distributed systems of sequential processes (no interrupts) with no system buffering.

Key Words and Phrases: interprocess communication, computer networks, ports.

CR Categories: 3.81, 4.32, 4.39

1. Introduction

The design of an interprocess communication mechanism (IPCM) usually starts with a description of the desired behavior of the system and the services to be provided. In selecting the features to be incorporated into the IPCM, the greatest amount of care is required, for these features are interdependent to a great degree, and it is crucial that the design process start with a complete, detailed specification of the system to be designed, with the consequences of each decision fully explored and understood. The temptation of piecemeal design is to be avoided at all costs.

The major aim of the paper is to point out the interdependence of the features to be incorporated in the system. In some cases, the incompatibility between certain features is direct and obvious. But often two features which look quite independent turn out to affect one another, and these are the more interesting cases. Unless the trade offs involved are explored at the outset, it is possible to find oneself 'locked out' of certain desirable features, because of unforeseen implications of an earlier decision. Though certain combinations of alternatives are outright incompatible, there are also cases where two features can both be accomodated, but at the expense of some basic design principle. This often results in some horrendous code 'patched' into the system, and much elegance is lost. The resulting system is harder to implement, verify, understand, debug, and maintain. These are the questions which extract a "well, we didn't actually implement it that way," response from system designers.

Unfortunately, with few exceptions [6, 10, 15], there is little guidance to be found in the published literature on this important point - how to arrive at a consistent and elegant design. This paper is a modest attempt to help fill this gap. The paper will address itself to general concepts rather than to the specifics of a particular design, although it was influenced to a considerable degree by the experience gained in the design and implementation of the Stony Brook System [2]. A brief description of this system is given next to provide a context for the discussion.

1.1 The Stony Brook System (SBS)

SBS is intended to function both as a stand alone system and as a node in a network. The aim was to obtain a design which was simple and elegant in both environments. Also the overhead due to the network operation was to be minimal.

In order to localize the effects of changes in the network, the Remote Executive (REX) is designed as a separate module and forms the sole interface to the network. The basic IPC is simple and supports only direct communication between local processes.

Processes wishing to communicate across the network cannot establish direct connections and

*This work was supported in part by the NSF Grant JO 42562.

†Present address: Dept. of Industrial Engineering
Texas A & M University
College Station, Texas 77843

must go through indirect paths. On the other hand users are primarily interested in the communication between the (logical) sender and receiver of a message.

In order to reconcile these seemingly contradictory goals (i.e. to have an IPCM which knows only about direct connections and users who are only interested in logical connections) SBS uses a strategy which amounts to building a network communication facility consisting of intermediate processes. These intermediate processes are not part of the basic IPC of any site. They are inserted in the communication path through the directory or broker process when a connection is set up. The intermediate processes are the only ones to know about the indirect nature of the communications which involve them. The key to making this strategy work is a judicious choice of the set of primitives of the simple local IPCM. For example, the basic system can only provide status information about the outcome of a direct transaction, whereas the user needs information about the logical message. To bridge this gap, the IPCM allows a 'delayed status return', which is used by the intermediate process to supply the status only when it finds out the ultimate outcome. The primitives used by REX and other system processes are also available to any user. In addition to keeping the system simple, this philosophy ensures a powerful and flexible communication facility.

1.2 Features to be Considered

We begin with an informal description of the major features to be discussed.

Ports Processes communicate through ports [3, 15] which may be thought of as abstract connections. A transaction takes place only when both parties indicate their willingness through a "rendezvous" at the port. (In some cases, the initialization of this procedure presents certain problems whose solution is not trivial.)

Sequential Processes In keeping with the modern trend [7, 11] we assume that all processes are sequential.

Messages In a network which consists of disparate machines it is very desirable to deal with messages at the logical level. It is then easier to ship a message across the boundary between two machines with different word sizes and differing resources. This may be achieved, for instance, as a series of partial transfers without taxing the capabilities of the smaller machines. The participants need only be awakened when the whole transaction is complete.

System Buffering As this is awkward to implement with certain architectures, (e.g. on the PDP-15 on which SBS was implemented each process has to run within a contiguous block of core) we shall confine ourselves to systems which do not provide this facility.

Time-Outs In practice, the system can only retain information for a limited time; undelivered or partially fulfilled messages are timed out and deleted. A subsidiary question is whether the system also forgets about the status of messages that are timed out. Time-outs affect each of these situations differently and we shall consider each case in its context. Also sequential processes require time-outs, otherwise they would be forced to wait forever for a message which was never sent - a situation which is likely to arise in a network environment.

Status Information One of the facilities provided by a well-designed IPCM is to return information to the participants of a transaction as to its outcome. This status return facility is quite burdensome, especially in computer networks, and it was proposed [14] to eliminate it altogether in such situations. Although this would result in considerable simplification, it can be shown (see appendix) that if the system itself did not provide this facility, there is theoretically no protocol that the users themselves may devise to fill this gap and totally eliminate their anxiety [1]. At any rate, much of this paper is devoted to the limitations of what the system itself can and cannot do in this respect.

Well-Known Ports Modern operating systems which provide sophisticated communication facilities, usually take advantage of this capability to implement certain system functions. This increases modularity since such functions (e.g. file directory) can then be prevented from being buried deep into the system and can operate (more or less) as the user processes. They are then easier to debug, modify and tune up. The problem is that such processes must participate in communications with many others without having prior knowledge of the identity of their partner. This communication occurs through well-known ports which must at all costs be protected from malfunction. Undebugged or malicious user processes have to be prevented from interfering with the operation of well-known ports.

Insertion Property Computer networks seldom maintain the same configuration over extended periods. If a user process has to resort to a different protocol with every change a program which ran one day may not run the next. It is therefore desirable to insulate user programs as much as possible from such variations. This increases portability and flexibility. In this paper we discuss an extreme approach, namely the insertion property, [4] which makes all intermediate processes inserted between the two main participants totally invisible to them - a useful feature to have in a network. A less extreme and more practical version of the insertion property is also discussed, where the aim is not so much to prevent a process from detecting the presence of an intermediate process, but to enable a process to operate in the same manner in either case.

2. Centralized vs. Distributed Systems

2.1 Centralized Communication Facility

A centralized facility is characterized by the presence of a single agent who has the complete state information pertinent to a communication. Further such an agent will be able to change the state of a system in a well-defined manner. For example the IPCM (which is the centralized agent) may match SEND and RECEIVE requests of two

processes, transfer the data between their buffers and provide appropriate status to both. Each of these processes is then assured that the status received by the other process was consistent with the information it received. Though it is possible that the IPCM may crash during the state change, in practice, it is the heart of an operating system and if it crashes there will be no further interest in the resulting state of the system. We shall ignore such cases in our discussion.

2.2 Distributed Communication Facility

A distributed facility is one in which there is no single agent who knows the complete state at any time. The IPCM is composed of several independent components which have to coordinate and exchange the parts of state information each has. As a consequence there is potential delay in effecting a global change. Further, if one of the components of a distributed facility crashes we shall still be interested in the activity of the rest of the components. As an example (Figure 1), consider the two processes P1 and P2 on two different machines communicating through a network.

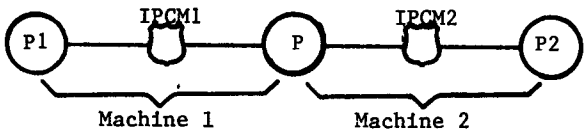

Figure 1.

The process P can be thought of as an interface between the two machines parts of which lie on each machine. The details of handling the network lines (which are not shown in the figure but are assumed to be absorbed in P) are managed by this interface. If one machine or a communication link crashes, we want the surviving IPCM's to continue their operation. At least one component should detect a failure and be able to communicate. (In the case of a communication link failure, both ends must know.) Note that a star configuration of many machines where the central node handles all inter-machine requests is distributed, assuming that the central node does not know the states of all processes on the peripheral nodes at any time.

Distributed communication can take place even in a stand-alone system if there are one or more intermediate processes taking part in a communication. The situation would be similar to Figure 1 except that since P, P1 and P2 are now all on the same machine, IPCM1 and IPCM2 are also the same. The main parties to the communication are P1 and P2, P being an intermediate process which performs some service, say translation and/or monitoring of messages in both directions. Each transaction between P1 and P2 consists of 2 steps (P1 to P and P to P2) which would normally be treated as independent transactions by the IPCM. For instance, the status returned to P1 would reflect only the outcome of the transaction between P1 and P although P1 is really interested in the eventual fate of its communication with P2. This means that if P1 (and P2) are to receive status information about the success/failure of the overall communication then P1, P2 and P must devise a fairly elaborate protocol. A solution which avoids these complications is the facility to ask for a delay in status as part of the RECEIVE primitive. This new RECEIVE & DELAY STATUS primitive has the effect that status return to the sender does not occur immediately upon the transmission but only when the receiver issues a SEND STATUS primitive. In the above example (assuming a message going from P1 to P2) P would use this facility to receive data from P1. It would then go ahead and relay the data to P2. Subsequently, when P itself obtains the status of this second step, it can issue the appropriate status to P1 through a SEND STATUS primitive.

2.3 Special Cases of Distributed Facility

We first eliminate a few pathological situations.

FACT 0: A perfectly reliable distributed system can be made to behave as a centralized system.

Intuitively, this is because the relevant state information which is distributed in several components is generally accessible. For the system to behave as a centralized system it is enough if the component IPCM at one end of a communication path knows the fate of a message at the other end. This can be achieved by an exchange of status information between the two IPCM's (through reliable communication).

In practice, it is unreasonable to expect perfect reliability of the communication links connecting the various components of the IPCM's. It is possible to relax this requirement, and we state without proof:

FACT 1: A distributed IPCM can be made to simulate a centralized system provided that
(1) the overall system remains connected at all times, and
(2) when a communication link fails, the component IPCM's that are connected to it know about it, and
(3) the mean time between two consecutive failures is large compared to the mean transaction time across the network.

In view of Fact 0, it is enough to show that reliable communication can be achieved under the above conditions. The informal justification is as follows: Link failure detection enables the nodes to adopt a scheme in which one and only one copy of an undelivered message is retained at any time. Thus an undelivered message cannot be lost and disappears from the network when delivered. Condition (1) ensures that there will always be a path from any node to another. A proper failure rate (condition (3)) together with the choice of a suitable routing strategy ensures that a message moving around within a subset of nodes in the network (while the target node is outside this subset) has to get out of the subset in finite time and this guarantees that the message eventually reaches the target. Precise bounds on the failure rate can be computed for any given routing strategy. One (rather inefficient) strategy, for

example, is that each time a message comes back to an intermediate node, it tries to send it through the oldest link on which the same message was sent before, so that each possible path is tried ultimately.

Both of the cases described above represent rather special situations because

(a) they involve inefficient and complicated algorithms,

(b) practical systems have other features such as time-out, which complicate matters, and

(c) the property of connectedness may be violated, in that the failure of a single link may result in two disconnected components.

In the following discussion, we consider distributed systems in general.

3. Status in Distributed Systems

3.1 Complete Status

Ideally, the status supplied to a process should specify completely the final outcome of a transaction (i.e. whether the message reached the destination). Such a status is called complete. If a system provides complete status to both processes, then the two parties not only know the ultimate fate of the transaction, but also know that they are in agreement as to what exactly happened. While it is possible to achieve complete status in a centralized system with reasonable assumptions, we now show that this is not the case in a distributed system where many compromises have to be made.

FACT 2: In an arbitrary distributed facility, it is impossible to provide complete status.

Two examples in which complete status cannot be provided are given below.

Assume first that condition (1) of FACT 1 is violated and at some time the network splits into two disjoint systems leaving two component IPCM's (IPCM1 and IPCM2) disconnected. If, at that time, IPCM1 was awaiting the outcome of a transaction (involving IPCM2) in progress, there is no way IPCM1 can provide complete status to the local party to the transaction.

Consider next a violation of condition (2) of FACT 1 and again a transaction involving IPCM1 and IPCM2 (Figure 2).

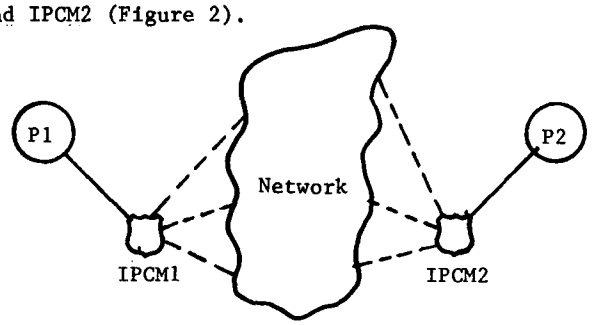

Figure 2.

IPCM1 cannot provide complete status to P1 until it is informed by IPCM2 about the fate of the transaction at the other end. But this status, supplied by IPCM2 itself uses unreliable communication paths. Therefore, IPCM2 cannot be sure that the status reached IPCM1, unless it gets an acknowledgement message, etc. This leads to an infinite exchange of messages between the two IPCM's. A more formal proof of this result is given in [1].

3.2 Time-outs

In practice unfulfilled requests of processes are timed out and the processes are notified. Furthermore, the system itself is usually limited by finite resources such as table space, etc., which hold control information in handling requests, so that it cannot afford deadlock situations which might arise as a result of a chain of requests waiting one for the other. A time-out mechanism is useful in breaking such chains. Next, we show that such considerations may prevent complete status, even if the system is totally reliable.

FACT 3: In a distributed system with time-outs, it is impossible to provide complete status (even if the system is absolutely reliable).

This can be shown by an example. Suppose (Figure 3) P1 sends some data to P2 through a chain of distributed components as shown. If after I1 has taken the data from P1 but before the status information has returned, P1's request times out, what status shall be returned to P1?

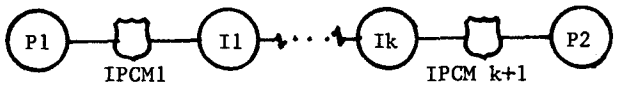

Figure 3.

IPCM1 has no knowledge of the final outcome of the transmission (the data may or may not have reached P2), indeed P2 may not be willing to receive it. Whatever status IPCM1 returns to P1 it may prove to be incorrect. Hence complete status to both parties is impossible. P1 has to be told of the uncertainty. Thus uncertain status is inevitable at times in a distributed facility with time-outs.

3.3 Insertion Property

An IPCM is said to possess the insertion property if it is possible to design an intermediate process P as in Figure 4b which remains invisible to P1 and P2; for any pair of primitives P1 and P2 issue, they should get the same status they would have obtained were they directly connected as in Figure 4a. Such intermediate processes are useful in network interfaces; they can do code translations etc., without bothering the two processes, P1 and P2.

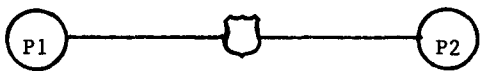

Figure 4a.

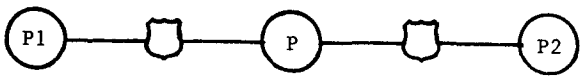

Figure 4b.

However the insertion property imposes additional constraints on the status that can be supplied as shown below.

FACT 4: In a distributed system with time-outs, the insertion property can be possessed only if the IPCM withholds some status information that is known to it.

To justify this we make the following observations. It is obvious that delayed status is necessary if the insertion property is to hold and the configurations in Figures 4a and 4b are to behave the same way. Consider the case of a message sent from P1 to P2. Once the data is read by P, P1 enters a state called await-status. If this await-status times out (before P could learn what happened to the data) what status can be provided to P1?

Clearly, he cannot be told of the eventual outcome since that information is not available yet. On the other hand we cannot very well tell him that he was awaiting status, which would imply that his message was received by someone. But what if the message never reaches as far as P2? This violates the insertion property since a comparable situation is not possible in Figure 4a.

If instead P1 was told that its original request was timed out, this again violates the insertion property because P2 may in fact have received the data and this does not happen in the situation of Figure 4a.

The only other way is to give an ambiguous status to P1, which leaves him in doubt as much as it would have, were P1 and P2 directly connected. One such scheme is to introduce a new status to cover the situation. Furthermore, this status must arise at least in one situation in which the two processes are directly connected. Thus, a deliberate suppression of what happened is introduced by providing the same status to cover a time-out which occurs while awaiting status and, say, a transmission error. If, in addition, it is stipulated that a RECEIVE request always delays status to the sender, then the insertion property may be achieved. Thus if P1 gets such an ambiguous status, he does not know whether a transmission failure occurred or his await-status timed out. Both these situations are possible in Figures 4a and 4b. In any case P1 has to conclude that P2 may or may not have received the data.

Thus the IPCM is forced to hide information on purpose to preserve the insertion property under the above conditions.

4. Logical and Physical Messages

The basic function of an IPCM is the transfer of data between two or more processes and the synchronization of those processes. To effect this synchronization, the data may be thought of as being divided into messages, regardless of whether the IPCM itself is "message oriented" or "connection oriented".

Because of various limitations imposed by either the IPCM or the programs themselves, it may be necessary to divide messages into several units each of which may be sent through the IPCM as the result of a single operation. The sizes of these units depend on the buffer sizes of the processes involved in communication.

4.1 Buffer Size Considerations

At a relatively early point in the design of any IPCM which provides no system buffering, a decision must be made as to the course to be followed in the case of unmatched buffer sizes, as shown in Figure 5.

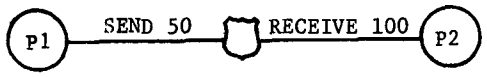

Figure 5.

The problem can of course be avoided by setting a system-wide standard buffer size. But this is too restrictive in a network of heterogeneous systems. If a mismatch is to be tolerated one approach is to satisfy the request with smaller buffer size and tell both parties what happened. This strategy is unattractive because it forces the processes to deal with the low level details of communication. (It also violates the insertion property.)

A more attractive solution is the design which allows for partial transfers. In such a design the information specified in the smaller request (50 words in the example) is transferred and only the process which issued the smaller request is awakened. The other process remains asleep awaiting further transfers. An end of message (EOM) indicator is also required to wake up the receiver even when its buffer is not full.

If a system is to support partial transfers, time-outs and the insertion property simultaneously there is a problem: suppose that the RECEIVE request in our example times out after receiving the first 50 words. Telling the process how much information is present in its buffer violates the insertion property so that we would have to return an uncertain status in this case. This is but one example of a situation which arises often in a system with partial transfers: a reasonable strategy would violate the insertion property by divulging the buffer size. We therefore propose a weak insertion property, where the availability of buffer size information is tolerated. This retains most of the advantages of the strict insertion property since only programs which explicitly attempt to detect the buffer size of their partners would be affected.

4.2 Partial Transfers and Well-Known Ports

Consider the situation shown in Figure 6.

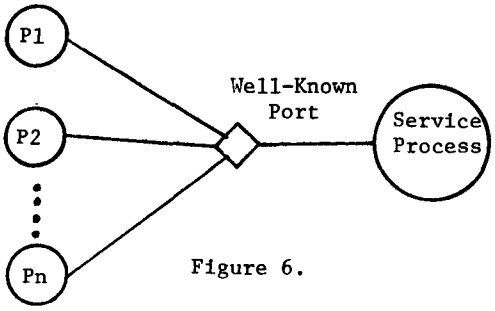

Figure 6.

A single service process is accepting requests from several different user processes (P1-Pn). The service process might be a directory process, a compiler, or any other process which provides a general service and accepts messages through a well-known port (one which is known to all processes without recourse to a directory or broker [5]; obviously, the broker process must have a well-known port). The example in Figure 6 demonstrates the problem.

If P1 (in Figure 6) sends a message that is not complete to the service process (the EOM indication is not ON) and does not fill the service process buffer, there are two problems we must consider.

First the port must be "reserved" for P1. We cannot allow a message from P2, for example, to be used to complete the RECEIVE request which was partially fulfilled by the partial message from P1. Secondly we must devise some method of handling the situation when the reservation times out, since there is no way to tell P1 that the first part of the message has timed out and thus been ignored. P1 is preparing to send the second part and is not listening for incoming messages from the service process. This means that the second part of the message may eventually arrive with the EOM indication set so that it looks like a complete message.

Since none of these problems arise in a system without partial transfers, another solution is to ban partial transfers to service processes with well-known ports. This is the approach ~~taken in ARPANET~~, ~~where~~ communication to well-known ports are restricted to short, complete messages [9] which are used to setup a separate connection for subsequent communication (see Figure 7).

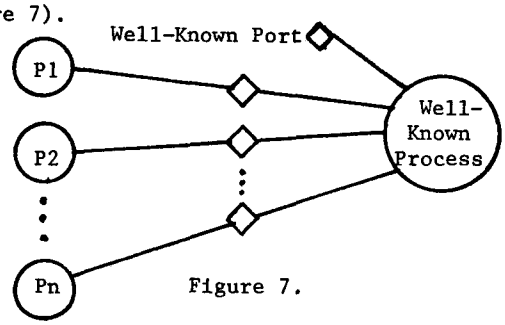

Figure 7.

It should also be noted that the problems with well-known ports arise through their interaction with time-outs and partial transfers. An IPCM without partial transfers will not have these problems since there is no time when only one party to the communication is awakened. True, the service process may not have the entire message, but it is awakened and may do whatever it pleases with the first part of the message (ignore it, buffer it, etc.); the burden is no longer on the IPCM.

4.3 Processes Using Many Ports

Consider the arrangement shown in Figure 8, where a server process accepts requests for service from many users. Such a situation actually came up in the design of SBS [2] where the REX was the service process. As all network communications go through REX it had to operate efficiently.

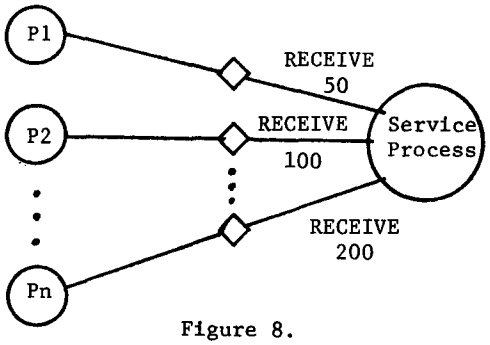

Figure 8.

First of all, some sort of primitive that checks several ports simultaneously is required - since we are examining systems where a process goes to sleep upon executing an IPC primitive. This primitive may actually perform the operation (it may act as a list of RECEIVE's with different ports and possibly different buffer areas) or it may be a QUERY operation, which does not transfer data, but awakens the service process when data is available on any of the ports (the service process may then issue a RECEIVE).

Also if the service process is to run efficiently it should not be made to wait on a partial transfer from one process while there is a complete request to be processed. The question once again is what to do with partial transfers. Suppose we had a SEND 25 from P1 (in Figure 8). If we put 25 words in a buffer in the service process, we are forcing the service process to have a separate buffer for each request. It we do not wish to make this requirement, what shall we do about the message? If we wake up the service process and tell it about the 25 words, the strict insertion property is violated (and so is the definition of partial transfers). If the service process remains asleep and awaits the rest of the message from P1, we would not be able to process a complete message which might arrive from P2, and would thus fail to process the first complete message to arrive.

Even if we allow separate buffers for separate ports, we have problems. If the 25 words from P1 are stored in a buffer while the service process

continues (asleep) to wait for messages from all ports, a complete message may come in from some other process. The service process must now be awakened, for there is a complete message awaiting its attention. If the service process is told about the partial message from P1, the strict insertion property will be violated. More importantly, the service process will be forced to implement internal buffering. On the other hand, if nothing is said to the service process about the partial message from P1, the IPCM would lose all record of the message without telling anyone, a poor feature to design into any IPCM.

One solution to this problem might be to forbid partial transfers in this situation. Another is to ban them altogether and use either the standard buffer size or the wake-up on matching scheme. However, if the network supports partial transfers at all, a service process like REX (which participates in every network communication) must also accept partial transfers. A solution with partial transfers is presented below.

4.4 Buffer Processes

This solution to the above problem assumes that the system supports dynamic process creation. In Figure 9 we show the modified configuration where the S_i act as buffer processes in the communication between the P_i and the service process.

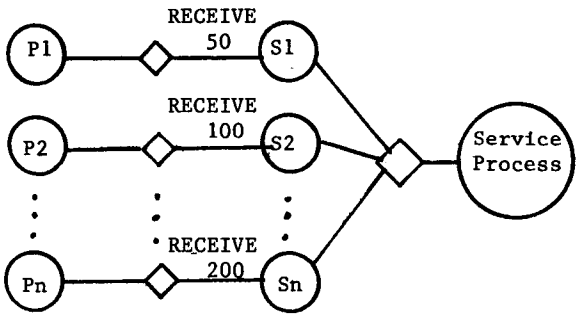

Figure 9.

Whenever a process P_i asks to be connected to the service process, a new S_i is created for it and the connection shown in Figure 9 is established. Each S_i accepts data from P_i until a logical message is complete, sleeping whenever necessary. It can then forward the complete message (with the EOM indication on) to the service process. All the messages received by the service process are complete since partial transfers are "filtered out" at the level of S_i.

With reference to the problem of partial transfers to well-known ports (section 4.2), we note that a similar solution is not possible there: the S_i can only be inserted when the connection is initialized but communication through well-known ports does not involve initialization. Also recall that the ARPANET solution to this problem creates the configuration shown in Figure 7. Although S_i may be inserted between P_i and the well-known process during the establishment of the separate connection, the original well-known port still remains (and cannot have partial transfers). At least one such well-known port, accessible without initialization, is inevitable in any system: this would belong to a directory or broker process through which all other connections are initialized.

5. Concluding Remarks

At several points in this discussion, we have stated that certain sets of properties do not go together, and have indicated modifications to eliminate the incompatibilities. It should be emphasized that these modifications by themselves do not guarantee realizability - they only indicate the existence of a suitable set of primitives to achieve the desired behavior. Also, the inclusion of other features, not discussed in this paper, may further complicate the problem.

In discussing status returned to the users, we have indicated how the presence of certain other features limits the information that can be provided. In fact, we have shown situations in which uncertain status had to be returned, providing almost no information as to the outcome of the transaction. Because of this, one might be tempted to design a system which provides no status at all. However, in a well-designed system with reasonable time-out intervals, etc., the above situations should not occur with any frequency, and it is still possible to provide meaningful status most of the time.

The insertion property, in its strict form, imposes far too many constraints and complicates the design of the system, so that it is difficult to make a case for it. On the other hand, the weaker version is relatively easy to incorporate; and has most of the benefits of the stronger version. In particular, it permits the design of user programs which are insensitive to changes in the environment.

Finally, we list a set of features which may be combined in a working IPCM.

(1) Time-outs
(2) Weak insertion property and partial transfer
(3) Buffer processes to allow
    a) many-to-one ports, and
    b) service processes using multiple, independent ports to handle requests arriving asynchronously
(4) Well-known ports - with appropriate methods to deal with partial transfers to them.

A similar set of features is designed into SBS.

APPENDIX: User Implemented Protocols

To show that no amount of user protocol can solve the problem in a manner to dissipate the anxiety of both parties as to the outcome of a transaction, consider the following model.

A group of gangsters are about to pull off a big job. The plan of action is prepared down to the last detail. Some of the men are holed up in a warehouse across town, awaiting precise instructions. It is absolutely essential that the two groups act with complete reliance on each other in executing the plan.

Of course, they will never get around to putting the plan into action, because the following sequence of events is bound to take place.

1. A messenger is dispatched across town, with instructions from the boss.
2. The messenger reaches his destination. At this point both parties know the plan of action. But the boss doesn't know that his message got through (muggings are a common occurrence). So the messenger is sent back, to confirm the message.
3. The messenger reaches the boss safely. Now, everybody knows the message got through. Of course, the men in the warehouse are not aware that step 3 occurred, and must be reassured. Off goes the messenger.
4. Now the men in the warehouse too know that step 3 was successful, but unless they communicate their awareness...

.....
.....

Note that the needs of both parties are quite reasonable. They simply want to reach a state where

(1) The original message (i.e., the plan of action) is successfully delivered, and
(2) Both parties know that they are in mutual agreement that (1) occurred.

Fact The sequence cannot terminate successfully.

Proof (a) Clearly the sequence contains at least one message of importance.

(b) Assume that it is possible to reach the desired state after a finite sequence of messages. Then there must exist a number $n \geq 1$ such that n is the length of the shortest sequence which gets us to this state. Since this is the shortest sequence, the last message in it is important: if the n'th message gets lost, the desired state cannot be reached. The sender of the n'th message must receive acknowledgment. This means that the sequence is at least of length $n + 1$. The assumption is contradicted and the sequence cannot be finite.

Note also that the sequence is infinite even when none of the messages are actually lost.

At first glance it would seem that if the two processes are in continuous communication, the problem can be solved by including a sequence number [8] as part of each message. But this is not really so: sequence numbers are analogous to the step numbers in the above example. At any time the process receiving the highest numbered message knows the complete state while the other lives in doubt. Thus in practice only sequential events can be controlled but simultaneity cannot be achieved by this means.

References

1. Akkoyunlu, E.A. "On the Limitations of Acknowledgement Messages". Working paper, ACM Interprocess Communications Workshop, March 1975.
2. Akkoyunlu, E.A., A. Bernstein, and R. Schantz "Interprocess Communication Facilities for Network Operating Systems". COMPUTER 7, 6 (June 1974).
3. Balzer, R.M. "Ports - A Method for Dynamic Interprocess Communication and Job Control". Proc. SJCC, Vol. 38, 1971.
4. Bernstein, A. and K. Ekanadham "Interprocess Communication in a Network". Network Systems and Software, INFOTECH State of the Art Report 24, INFOTECH Information Ltd., 1975.
5. Bressler, R., D. Murphy, and D. Walden "A Proposed Experiment with a Message Switching Protocol". NIC #9926, May 1972 (available from Network Information Center, SRI, Menlo Park, California).
6. Brinch-Hansen, P. "The Nucleus of a Multiprogramming System". CACM 13, 4 (April 1970).
7. Brinch-Hansen, P. "An Approach to Multiprogramming ". Information Science, California Institute of Technology, Pasadena, March 1973.
8. Cerf, V.G. and R.E. Kahn "A Protocol for Packet Network Intercommunication". IEEE Transactions on Communications, Vol. COM-22, No. 5, May 1974.
9. Crocker, S.D., J. Heafner, J. Metcalfe and J. Postel "Function-oriented protocols for the ARPA computer network". Proc. SJCC, Vol. 40, 1972.
10. Danthine, A. and J. Bremer "Définition, Représentation et Simulation de Protocols dans un Contexte Réseaux". Journ. Intern. Mini-ordinateurs et Trans. de données, AIM, Liege, January 1975.
11. Goos, G. "Communication in process structures" Technical University of Munich, 1972.
12. Haberman, N. and A. Jones "Interprocess Communication Mechanism". Network Memo, Department of Computer Science, Carnegie-Mellon University.
13. Metcalfe, R.M. "Strategies for Interprocess Communication in a Distributed Computer System" Proceedings of the Symposium on Computer Communications and Teletraffic, Polytechnic Institute of Brooklyn, April 1972.
14. Walden, D. Private Communication, 1973.
15. Walden, D. "A System for Interprocess Communication in a Resource Sharing Computer Network" CACM, 15, 4 (April 1972).