Project MAC June 17, 1975

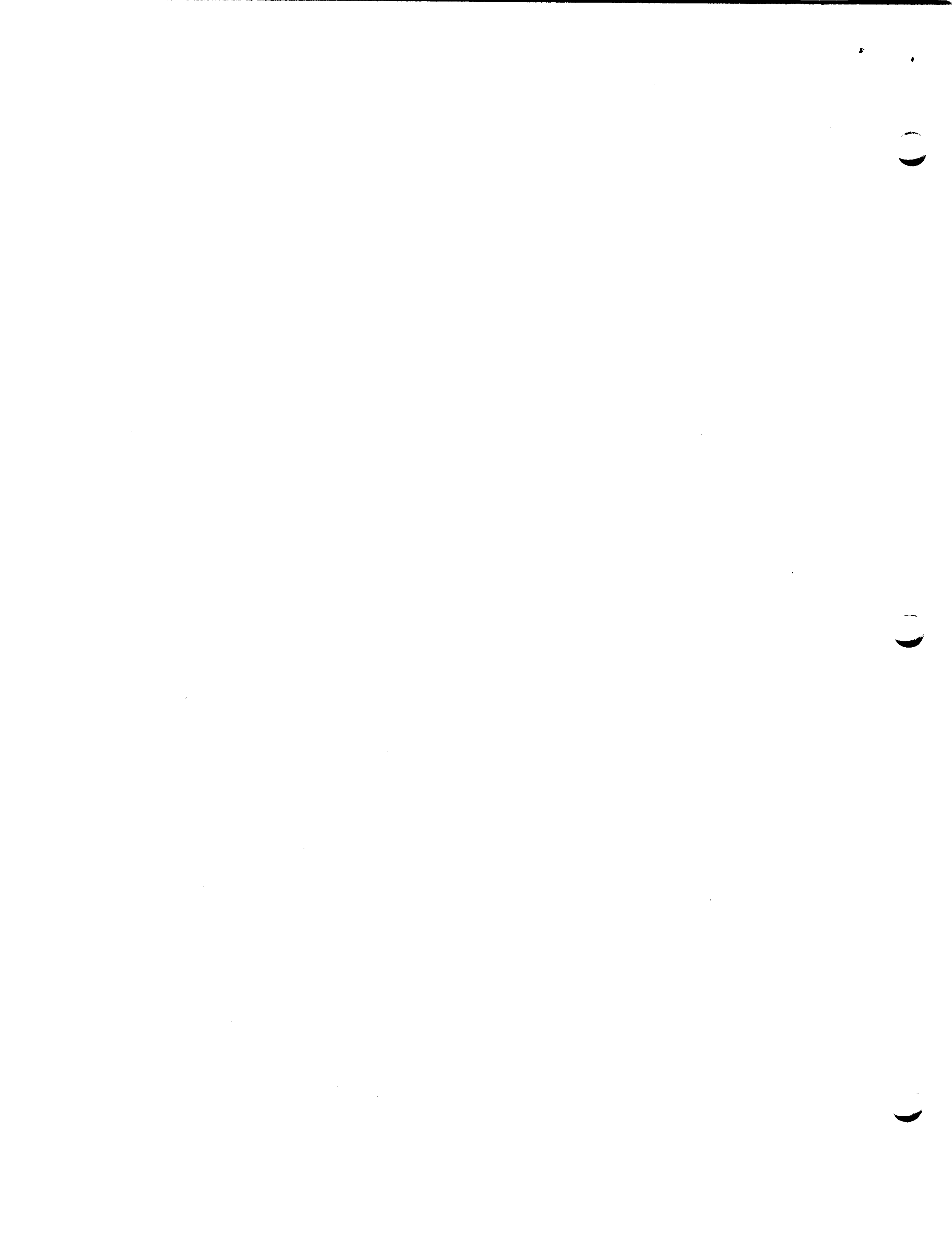
Computer Systems Research Division Request for Comments No. 82

Some Comments on the Procedure Call Protocol

by Raj Kanodia

The enclosed paper was distributed at the National Software Works (NSW) Working Group meeting held at Stanford Research Institute during July 11-13, 1975. The Procedure Call Protocol (PCP) is being developed at SRI to be used for constructing NSW. The PCP is an interprocess protocol that permits a collection of processes within one or more ARPANET hosts to communicate with each other via procedure calls. Its intent is to make it easy to use remote procedures. It is likely that PCP will be redesigned in the near future. If you would like to provide some input for the redesign effort by way of suggestions, comments, or constructive criticism, please communicate it to me.

This note is an informal working paper of the Project MAC Computer Systems Research Division. It should not be reproduced without the author's permission, and it should not be referenced in other publications.



Some Comments on the Procedure Call Protocol

by Raj Kanodia, Project MAC, M.I.T.

> *"There are philosophers to whom a single departure from the norms of common sense acts only as a stimulus to further more exciting philosophical adventure in the realm of speculation, but, I confess that, for my part, I regard such a departure rather as a danger signal, warning that it would be wise to consider whether the steps which have led to this departure are as secure as they appear to be."*
>
> *- Winston H. F. Barns*

The purpose of this note is to provide further material for the discussion on the Procedure Call Protocol (PCP) started by Rick Schantz in the NWG/RFC #684 [Schantz]. Let us say this at the outset that because PCP has sufferred for not having received sufficient exposure in the early design stage (unlike other network protocols which were designed by network wide committees), this note, in attempting to correct that deficiency, might seem overly critical to some readers. However, our purpose in producing this note is merely to raise questions regarding PCP that we feel should be given proper consideration now. It is likely that an initial version of PCP will not (and can not be expected to) resolve all the issues that one might raise; however we fear that unless some of the problems were given proper consideration now, it may become impossible to provide solutions for them at a later stage. To the extent that some of the questions raised in this note may be due to some misunderstanding of PCP on our part, we hope that this note will start a dialogue among the concerned parties which will correct such misunderstandings.

Procedure Calling

> *"The existence of procedures goes quite far toward capturing the meaning of abstraction."*
>
> *- Liskov and Zilles*

Rick Schantz [Schantz] has argued that "procedure call protocol should not be the basis for multi-machine interactions", rather "a request and reply protocol along with suitably manipulated communication paths between processes forms a model better suited to the situation in which the network places us." We believe the issue is not one protocol versus another, we need protocols of both kinds. At present PCP and its related protocols lack suitable inter-process communication (ipc) and synchronization

primitives that could facilitate multi-machine interactions; however, such primitives can only complement the PCP rather than replace it. Later on in this note we shall say more about the need for ipc and synchronization primitives. In this section we shall attempt to demonstrate that under certain circumstances, remote procedure calling is an appropriate and even desirable method of utilizing resources in remote computer systems.

PCP incorporates notions of processes, interconnections among processes, procedures and procedure calls. The concepts of a process and interconnection among processes have existed for quite some time now and are embodied in earlier network protocols (TELNET, File Transfer, Host-Host). Procedures and procedure calling are fundamental to computing and are used in practically every computer. The most radical departure of PCP from the concepts that we are already familiar with, comes in the definition of inter-process procedure calls (or remote procedure calling) by which a process can invoke a procedure in a different process possibly in a different machine. The real issue, therefore, is whether remote procedure calling is an appropriate method of building distributed computation systems.

Let us first consider remote procedure calling with the following restrictions:

1 - All input arguments to the remote procedure and all input arguments from the remote procedure are values.

2 - The remote procedure has no side effects. In other words, the remote procedure neither alters an existing environment nor leaves behind a new environment after its return.

3 - The remote procedure does not depend upon any environment that is not an integral part of itself.

Notice that these restrictions do not preclude creation of a temporary environment which may be used during the activation of the procedure and which is destroyed at the time of return from the procedure. Most mathematical functions can be computed by remote procedures which execute under these restrictions. A procedure with these properties is, in essence, a data flow procedure [Dennis] and remote procedure calling is an appropriate method of invoking such a computation. The remote process is being treated as raw computing computing power and any number of procedures of this type can execute simultaneously (even in a single process) since they can not interact with each other. For such a computation, we might even choose to ignore the matter of recovery from network and host system failures since the computation is restartable.

When we attempt to relax some of the restrictions on the remote procedure calls, we run into difficulties. For example, if we

allow the arguments to remote procedures to be names of shared files then remote procedures might need to lock the files. Whenever there are locks, there is the possibility of dead-locks. Who is responsible for ensuring that dead locks do not occur? The callee or the caller? However, problems of this nature are not peculiar to remote procedure calling; they exist in all computation, local or remote. It really is a matter of good system design versus poor. If a procedure implements an easily understood and properly defined abstraction, then remote use of the procedure will cause very little difficulty.

There are certain computations for which the remote procedure call model is not appropriate. Most interactive computations fall into this category. Let us consider the example of a text editor. In most systems a text editor can be invoked either by a procedure call or as a command. From then on editor accepts a series of requests over a communication path (which might have been specified as a parameter to the editor command or procedure) and performs requested operations after ensuring that certain requirements have been met. (For example, one may quit out of an editor only after having saved a copy of the updated file.) It would be quite unsuitable to express each editor request in terms of one or more remote procedure calls. In such computations, there are at least two reasons why the request model is to be preferred over the remote procedure call model. First of these is that some information regarding the state of the computation is directly expressed in the execution state of the process in the request model. For example, in the request model one can not force the editor to perform two simultaneous "substitute" operations; it is implicit that the editor performs only one operation at a time. Whereas, in the remote procedure call model, information of this nature must be expressed in an environment as bit patterns and each procedure call must explicitly examine the environment to ensure that necessary constraints on the sequence of operations etc. are being met. In the remote procedure call model, one would need some sort of lock to ensure that only one "substitute" operation is performed at any given time on some file. Secondly, in the remote procedure call model, one must devise some method of specifying the environment that is passed from one procedure to another. In the case of an editor, this environment is likely to be host specific, however, a remote host must specify this environment in each call to perform some operation on the file.

Inter-process Communication and Synchronization

Processes executing in parallel require some kind of inter-process communication mechanism or synchronization primitives to coordinate their activities, for example Multics event channels [Spier], P and V operations on semaphores [Dijkstra], and block and wakeup primitives [Saltzer]. However,

PCP and its related protocols do not provide suitable primitives that will enable parallel processes to coordinate their activities. We illustrate the need for synchronization primitives by an example from the Process Management Package (PMP) which uses PCP to implement some of its functions. The PMP procedure CRTPHYCHN (Create Physical Channel) creates a PCP communication channel between two remote processes. CRTPHYCHN begins execution by allocating physical ports in the remote processes and then it initiates execution of the CRTPHYCHNEND in one remote process with a "passive" argument value (which implies that the remote process should listen for a request-for-connection (RFC)) and then it initiates execution of CRTPHYCHNEND in the other remote process with an "active" argument value (which implies that this remote process should transmit a RFC) [See PMP Version 2, page 12]. The sequence of actions is illustrated in figure 1 with process P0 executing CRTPHYCHN to establish a PCP channel between processes P1 and P2. P1 is the passive process and P2 is the active process. As shown in figure 1, processes P1 and P2 require synchronization at points A and B. Essentially, process P2 must not initiate transmission of RFCs until process P1 has initiated the action of listening for the RFCs from P2. The present implementation of PMP ignores this issue completely.

Presumably one could provide the required synchronization by splitting the CRTPHYCHNEND in process P1 into two procedures, CRTPHYCHNEND-part1 and CRTPHYCHNEND-part2 such that part1 returns to the caller after initiating the action of listening for the RFCs and part2 completes the connection when the RFCs arrive. Now, process P0 will execute part1 in P1 and only upon return from part1 in P1 will it initiate execution of CRTPHYCHNEND in P2 (see figure 2).

The method, just outlined, will accomplish the required synchronization. However, it is fairly clumsy and it does not properly represent the sequence of computation in various processes in a clear fashion, particularly in process P1. Essentially, what would otherwise be a sequential computation, has been artificially split into two parts. Part2 executes in an environment which has been left over from part1. Furthermore, it is now the responsibility of process P0 to ensure that part2 is executed only after part1 has been executed succesfully.

The above example illustrates a case requiring very simple synchronization. With the development of new programs for NSW, undoubtedly there will arise more complex situations requiring multiple synchronization points. Without proper interprocess communication and synchronization facilities, it may become practically impossible to handle these situations in an orderly fashion.

Multiple processors in a PCP process

> *"A process is a locus of control within an instruction sequence."*
>
> *- Dennis and VanHorn*

> *"A processor is an entity which performs transformation of information. A process is an entity which can control and define a virtual processor."*
>
> *- R. W. Watson*

As presently defined, a PCP process can have multiple processors in it.[1] The concept of a process was invented to permit multiplexing of a resource (the CPU) among several users. Since management of this multiplexing was the function of the operating system, a user was freed from the problems of managing this multiplexing. Therefore, it seems that introducing multiple processors in a single process will require a PCP process to duplicate the function that is already being performed by the operating system, namely, multiplexing of CPU resource allocated to the process among several processors. If the object of multiple processors is to enable simultaneous execution of two or more procedures in the sense that each procedure is guaranteed to get some finite percentage of CPU time allocated to the process then it should be pointed out that most systems will have to implement a PCP processor in terms of a full fledged process. While this implementation is not going to be any more efficient than creating a completely separate process in the first place, it has the disadvantage that all communication to the process pretending to be a PCP processor must be via the parent process. However, if the intent of having multiple processors is to permit apparently simultaneous execution of several coroutines[2] then a better method of achieving this goal would be to provide primitives for coroutine calls and their synchronization as is done in SIMULA [Dahl].

However, all such arguments aside, there are many practical problems associated with the concept of multiple processors. A point by point discussion of these problems follows.

---

[1] This issue was raised by Rick Shantz and Bob Thomas in their critique of PCP and NSW (March 4, 1975).

[2] If there are several coroutines in a process then at most one of them can be executing at any given time; other coroutines would be in a suspended state waiting for some event to happen which may cause or coincide with suspension of the executing coroutine and resumption of one of the suspended coroutines.

1 - Even though PCP permits a PCP process to deny allocation of more than one processor, the notion that multiple processors will be available has creeped into the design of PCP in a rather subtle way. The only suitable model for handling INS interrupts associated with PCP messages is to have a processor whose sole purpose is to listen to all the incoming channels and process the requests immediately upon arrival. This requires that there be at least one other processor for executing the user procedures. We shall discuss this issue at length in another section of this paper.

2 - It is likely that many computation systems built upon PCP will assume availability of more than one processor and will fail to work with hosts that do not provide multiple processors in a single process. In some sense this has already happened in the Process Management Package (PMP). Consider the following scenario. A process A is connected to processes B and C such that B and C are not connected to each other. Let C be a single processor process, executing some procedure on behalf of another process D. Now, if A invokes the PMP primitive ITDPRC (introduce-processes) to introduce B and C, the ITDPRC will attempt to call PMP primitives in processes B and C via PCP. But C will not be able to execute any PMP primitives on behalf of A because its single processor is already busy. As a result the ITDPRC in process A will fail.

The problem here is that even though succesful execution of ITDPRC requires availability of a processor in each of the processes being introduced, this requirement is not stated anywhere in the definition of ITDPRC. It is our contention that problems of this kind are not going to disappear even if the definition of ITDPRC (and other such PMP primitives) were to be changed to completely specify all the requirements for their succesful execution. In the first place, such specification will be hopelessly complicated and rather than facilitate use of PMP, it will only hinder it. Secondly, if we are ourselves lax in defining a basic protocol primitive, we can hardly expect that the environmental requirements of facilities built upon this protocol will be adequately described.

3 - In most systems, procedures execute in an environment associated with the process. Introducing the notion of multiple processors in one process immediately raises the question: "Is there a distinct environment with each processor?" For example, the internal static variables of a PL/1 procedure (<u>own</u> variables in ALGOL 60) in Multics are initialized when the procedure is

executed for the first time in the process. The question is: "Will a procedure executing on two different processors in the same PCP process have two different sets of internal static variables?"

Regardless of how the questions raised above are answered, each answer raises further questions. If we assume that there is only one environment associated with a multiple processor process, then each procedure must be prepared for simultaneous invocation in a single process, and it must synchronize use of all its internal data bases. If the answer is that each processor has its own environment then the processors must be dedicated for certain tasks and may not be interchangeable from one task to another as the PCP seems to imply. (See PCP Version 2, page 18).

4 - Management of processors: the above discussion essentially leads to the issue of management of processors. A process is after all a virtual resource created by multiplexing basic resources (i.e. the cpu, memory, etc.) which are managed by the operating system. Multiplexing this virtual resource (the process) further requires another level of management. It needs to be clearly spelled out exactly how the processors are managed; whether locally or by the remote processes.

It seems to us that in the definition of processors, many unstated assumptions have been made which are characterstics of a particular host system, namely TENEX. From the published documentation regarding PCP, it is not clear to us what the motivation is for introducing the notion of multiple processors in the PCP process. If the concept of multiple processors is retained, then we suggest that this problem be viewed in more abstract terms rather than in terms of a particular system. This will help in clearly identifying the relevant issues and prevent unstated, undocumented assumptions from creeping into the PCP which could, otherwise, become potential pit falls.

Even though we might implement multiple processors in a Multics PCP process, we prefer that a PCP process be defined to have only one processor and simultaneity be achieved by creating multiple processes. This will help keep PCP small and manageable, and will avoid duplication of functionality which is already provided by the operating systems.

Interrupts with PCP Messages

> *"The one capability of commonly described operating system psuedo-processors which is not included in our psuedo-processor is the "interrupt," or "courtesy call," a jump to a special subroutine in response to an arbitrarily timed (asynchronous) signal, for example, from an input/output channel."*
>
> *- J. H. Saltzer*

A PCP process can send two types of messages to another PCP process known to it, the so called IPC messages (CRTPRC, DELPRC, etc.) and PCP messages (CALPRO, ABRPRO, INTPRO, etc.) The IPC messages may be sent only to direct inferiors and may be viewed as exercising the ultimate authority over a process. For example, a request to terminate a process (DELPRC) will be acted upon regardless of whether the inferior process is busy or not. On the other hand PCP messages are merely requests to a process to take some action on behalf of the sender process. The PCP message CALPRO requests the receiving process to execute a local procedure if a processor is available. The way PCP is defined now, arrival of each PCP message is accompanied with an INS interrupt, the receiving process is expected to process a PCP request immediately upon its arrival, regardless of whether all its processors are busy or not, and if it can not honor the request, it must send a reply to this effect.

This behavior of PCP can not be suitably modeled unless every PCP process has at least two processors; one dedicated to the task of processing PCP messages and another for executing procedures requested by remote processes. It seems to us that this peculiar behavior accompanying PCP messages is not likely to serve any useful purpose. If an inferior process is executing a procedure on behalf of its superior then, presumably, the superior is aware of this fact and will not make any more requests until the previous one has been completed. If the inferior process is being simultaneously used by a third party process, then either the third party process should coordinate its activity with the superior or it should be left to the inferior process to determine the manner in which the requests from multiple processes are to be honored. Our point is that in a properly programmed system a process will not make a request upon a remote process and expect it to be acted upon immediately unless it has made some previous arrangement for resources. Suppose the inferior denies a request from the third party process, now what does the third party process do? Does it keep poking the inferior in the hope that some day its request will be honored? Or does it abort computation and return an error condition? Rather than keep poking, the process should make a request that will be honored when the resources become available. In the later situation (abort computation and return error) the system

is not well programmed and there should be better management of processors. In any case, interrupts associated with PCP messages can be eliminated. The requests will be automatically queued in the channel and PCP will act upon them according to availability of processors. This will also eliminate the need to have a processor solely dedicated to processing of PCP messages. The two exceptions to this rule are INTPRO and ABRPRO messages which will still cause interrupts. However, their arrival implies that execution of some procedure be interrupted which frees a processor which can be switched to processing of these messages.

The issue of interrupts with PCP messages strongly interacts with the multiplicity of paths from which a process can receive requests. If all remote processes coordinate the use of resources in a given process then requests coming from multiple paths present no problem. However, uncoordinated use of a process should be permitted only if this process implements a service which is uniformly available to many other processes. This service can be accurately modeled as one process listening to requests on many connections. This process may either perform the request itself or delegate it to some other process under its control. The point is that details of the implementation of the service are imbedded in the procedures that define the service. Users of the service need not worry about management of resources within the service.

Reliability issues

> *"If a piece of software meets its specification, it is correct - if it does not, it is incorrect. Reliability, in contrast, is a statistical measure relating a system to the pattern of demands we make upon it. We consider a system to be highly reliable, if it highly probable that, when we demand a service from the system, it will perform to our satisfaction."*
> *- D.L. Parnas*

A distributed computation system utilizes resources that are autonomously and independently controlled and that often have reliability and availability problems. When the number of components involved becomes large, the reliability problem for the system as a whole becomes severe. It is, therefore, fair to assume that a distributed computation system should not only be cognizant of the fact that individual components may break down but it should also specify recovery measures in case of failure of one or more components rather than abort the entire computation. Our point is this: a protocol designed to facilitate construction of computation spanning machine boundaries and utilizing a network as the basic means of communication, must take into account the fact that at times ARPA

Network connections do break down and that the host systems do crash. This protocol should provide for the following:

1 - Means of specifying action to be taken in case of failure. For example, when a process creates an inferior process, it should be able to specify exactly what will happen to the inferior process if the communication path between them breaks down or the host upon which the parent process is located, breaks down. In fact, for the sake of consistency, specification of such action should be mandatory and be an integral part of creating a new process. If it were not so, then each host will apply its own recovery mechanisms (which are likely to be different from host to host) and there would be chaos.

2 - Mechanisms to facilitate recovery from a failure. This point can be best illustrated with the classic example of a user logged into a system from a terminal. If some how, the terminal hangs up, the system could either destroy the user process or simply stop it for a brief period of time during which the user has the option of "redialing" to his suspended process and thus re-establishing the connection. This "dial-up" mechanism facilitates recovery from a failure. The ATTACH primitive of the Process Management Package (PMP) can provide recovery under some circumstances, but this whole issue needs to be carefully thought out.

Logical Channels in PCP

When a process introduces two other processes via the ITDPRC primitive of the PMP, PMP creates a logical channel between the two processes. This logical channel uses the physical channels between the introducing process and the ones being introduced. The ITDPRC also permits its caller to specify creation of a physical channel (in addition to the logical channel) between the processes being introduced. Furthermore, a logical channel is not used if there exists an associated physical channel. Therefore, logical channels are unnecessary. Communication via physical channels is quicker and more efficient than via logical channels. The kind of complexity that this mechanism introduces into PCP can be best illustrated by considering a logical channel between a TENEX (called TENEX-A) and non-TENEX system going through another TENEX system (called TENEX-B). Now a message sent from non-TENEX system to TENEX-A will be sent to TENEX-B in 8-bit ASCII. Upon receipt of this message, the intermediate host, TENEX-B will have to translate the message into 36 bit format to transmit it to the destination, the TENEX-A. If the logical channels are cascaded, it is possible that message will have to be converted and reconverted from one format into another

in several intermediate hosts. A transmission via a logical channel requires at least two network transmissions instead of the one required for physical channels.

We suggest that the ITDPRC primitive always create a physical channel and that logical channels be completely eliminated.

Acknowledgements

Many of the ideas contained in this note resulted from several hours of discussion with Doug Wells. The need for interprocess communication and network reliability considerations has already been emphasized by Richard Schantz [Schantz]. Dave Clark and Doug Wells carefully read a draft of this commentary and provided helpful comments. The author gratefully acknowledges the help of Jon Postel and Jim White who have promptly responded to all my queries regarding PCP and explained its intricacies.

References

Dahl, O.J. and Hoare, C.A.R. *Hierarchical Program Structures*
3rd monograph in *Structured Programming*
Academic Press, London and New York, 1972

Dennis, J.B. *First version of a data flow procedure language*
MAC Technical Memorandum 61, Project MAC, M.I.T.,
Cambridge, Massacusetts, May 1975

Dennis, J.B. and Van Horn, E.C.
Programming Symantics for Multiprogrammed Computations
Communications of ACM, volume 9, number 3, March 1966,
pp 143-155

Dijkstra, E.W. *The structure of THE multiprogramming system*
Communications of ACM, volume 11, number 5,
pp 341-364, May 1968

Liskov, B. and Zilles, S. *Programming with Abstract Data Types*
Proceedings of ACM SIGPLAN Conference on Very High
Level Languages, SIGPLAN Notices volume 9, number 4,
April 1974, pp 50-59

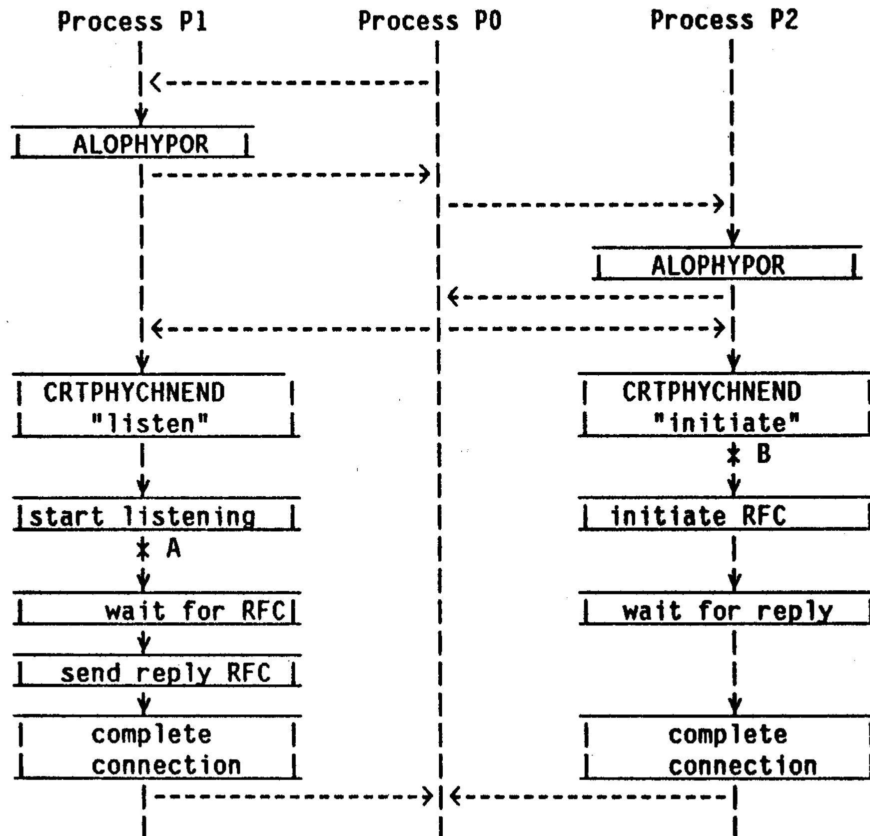
Parnas, D.L. *The influence of software structure on reliability*
Proceedings, 1975 International Conference on Reliable
Software, pp 358-362, April 1975

Saltzer, J.H. *Traffic Control in a Multiplexed Computer System*
MAC-TR-30 (Thesis), Project MAC, M.I.T.,
Cambridge, Massachusetts, July 1966

Schantz, R. *A Commentary on Procedure Calling as a Network Protocol*
NWG/RFC 684, NIC 32252, April 1975

Spier, M.J. and Organick, E.I.
The Multics Interprocess Communication Facility
Second Symposium on Operating System Principles,
Princeton University, October 1969

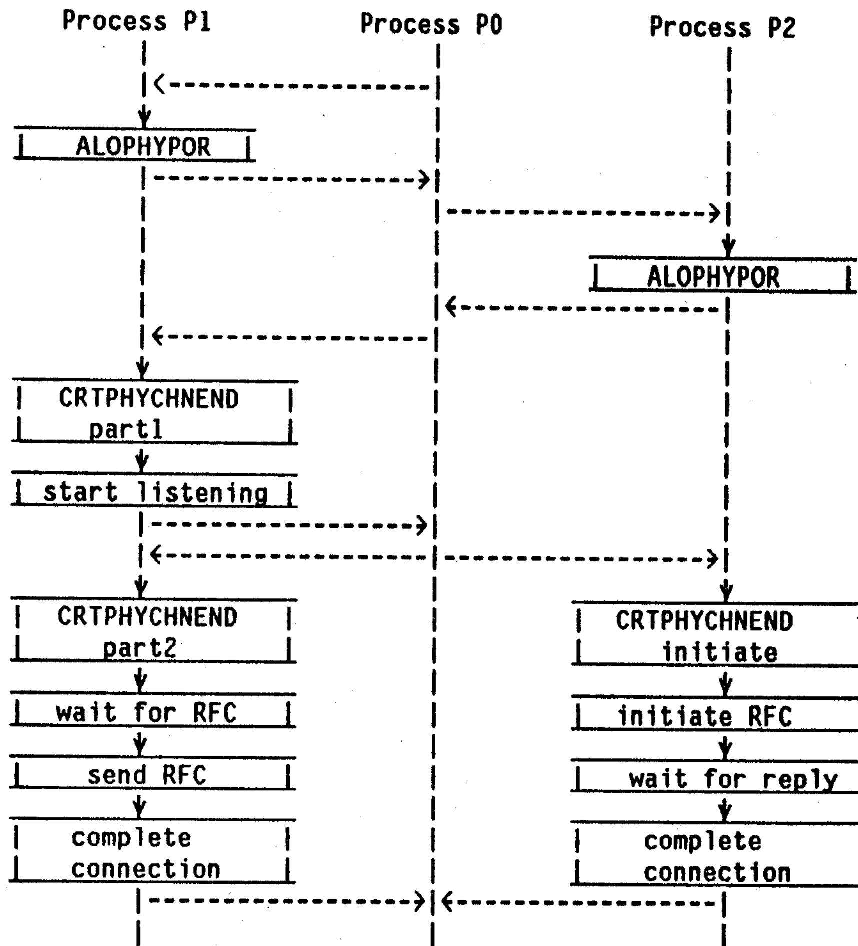
Watson, R.W. *Timesharing System Design Concepts*
McGraw-Hill, New York, 1970



PMP procedure ITDPRC in process P0

NOTE: Upon reaching point B, process P2 should continue only if process P1 has already reached point A.

Figure 1



PMP procedure ITDPRC in process P0

NOTE: Synchronization is achieved by splitting CRTPHYCHNEND in P1 into two parts. Return from part1 provides the synchronization point to process P0 which then initiates activity in process P2.

Figure 2