M.I.T. LABORATORY FOR COMPUTER SCIENCE May 21, 1982

Computer Systems Research Request for Comments No. 224

Considerations of Message Based System Design

by David D. Clark

The attached document was recently written by David Clark at the Computer Laboratory in Cambridge, England.

[illegible]

COMPUTER LABORATORY, CAMBRIDGE

Systems Research Group Note

David D. Clark - 10 May 1982

Considerations of Message Based System Design

1. Introduction

A number of operating systems have been built in which processes communicate with each other and/or the supervisor by sending and receiving interprocess messages. Dispite this, there remains considerable uncertainty as to what the details of message passing semantics should be. This note discusses some specific observations about message passing and related system issues, which arose during the implementation of network protocol software on a message based operating system, Tripos. The note suggests some directions for future operating system design.

Network protocols provide a hard test for message passing systems, because protocols are naturally structured as a number of cooperating asynchronous processes that demand very high efficiency in their interprocess communication. A typical protocol module, running on a minicomputer of average speed, may require only a millisecond or so to complete its actions, after which another process must be invoked to carry on. Since sending a message in many systems can take longer than that, the overhead of message passing can become the limiting factor in good overall performance. Further, protocol modules tend to interact in complex ways, which are often not properly supported by the system. This can cause what might have been a clean structure of many processes to get very messy.

This note discusses three aspects of system support for network software: the detailed semantics of message passing, the scheduling mechanism required, and the storage requirements for processes. Each of these points relates to the way messages are used in the system, and to the way that the basic structure of the system supports the intended applications.

It will be argued that the features required to support network protocols are in fact general requirements, which should influence the design of new systems for a variety of purposes.

2. Message Passing Semantics

The most basic argument about messages is whether every message must have a reply. One version of message passing has the sending process, on sending a message, halt pending a reply. Other versions allow the sender to continue, and to await the reply whenever convenient. Finally, some systems have no concept of reply, and view each message as a separate entity.

The argument between these is often based on what the perceived need is. Within limits, one structure can be made to resemble another. Obviously, the pattern of message and reply can be adapted by the application, even if the system itself does not support it. On the other hand, if the system requires a synchronised reply, the application can provide a separate process to await it, and continue computing within other processes. What is missing from most of these arguments is actual experience.

This particular implementation project provided experience in the specific domain of network protocols. In this case, it is clear that the send-reply pattern is unneeded and unwanted.

There are several problems with the send-reply pattern. First, in many cases, perhaps the majority, there is no natural use for the reply. One process is telling another that it can take over the processing, and the first process then goes on to something else, giving no more attention to the matter of that message. A good analogy for this program structure is the assembly line, with each process doing its part and handing on the work. While the sending process could be built to expect the reply, it would represent unneeded work for that process. That extra work is especially undesirable because it is very costly. The reply may come back at a time when the first process is suspended, so it must be scheduled just to discard the reply. In a system in which this cost is important, then, message replys cause a significant overhead.

The idea that a special process should be created to send the message and receive the reply is not reasonable. Once that separate process has been created, messages must now be used to talk to it. The send-reply pattern thus continues to exist, but with the added confusion and overhead of an extra process in the way.

In fact, the send-reply pattern sometimes comes into existence, not to support an imagined application need, but because it solves a system level problem, which is the creation and deletion of the storage of messages. If the sender creates a message and the receiver destroys it, then the message must be created in an area in which the receiver also has modify rights. In general, this implies that messages are created in some sort of kernel area. This makes messages a shared system resource, with all of the obvious management problems. In particular, it is now possible for one malformed process to claim all of the message storage, and crash the system. It would be much simpler if each process created its messages in some area local to itself. But this usually means that the receiver is not in a position to delete the message. Hence the reply action, which has the nice side effect of returning the actual message to the process that can destroy it.

This approach seems weak. For this application, as was observed above, having to deal with unwanted replies was a real burden. More generally, what an operating system does is manage shared resouces. It should not be unwilling to take on the task of managing messages, especially if they are the basic mechanism of the system semantics. Later, this note will present a specific proposal for system level management of message storage.

There are a number of other problems with send-reply semantics, but more background is required to discuss them properly. Thus, it is time to consider some other aspects of message passing.

Waiting for a message is more complex than sending a message, because of the uncertainty as to which message may actually arrive. There are two general possibilities. First, the process can wait for a specified message, and ask the system to hold any others that may arrive, or it can wait for anything and sort what arrived on its own. Again, arguments are often based on what the user wants. Waiting for any message places a burden on the process, for the process must essentially create a private message handling mechanism which takes unexpected messages and sets them aside for later processing. The idea of specifying which message is next due attempts to put this burden on the system instead of the application. Unfortunately, this does not work completely, because the process may need to deal with whichever message comes first. If the system will let a process specify more than one message for which it is prepared to wait, this uncertainty can be handled, but now the process is again doing some of the message sorting, which was the burden the system was trying to take over in the first place.

This protocol project suggests strongly that waiting for any unspecified message is the normal and desirable form. Most strongly, it suggests that being able to wait only for one specific message in totally unworkable. Several systems, including popular ones like Unix, have had this defect. The evidence is quite strong now that a process must be able to wait for any one of several events, whether or not these must be enumerated.

When a system allows each process to wait for only one message, the usual application structure suggested is to create several processes, each to wait for one event. In fact, in this project, this structure was most natural. In normal operation, each process was waiting for one specific message at any one time. For example, when a packet was send, one process waited for a reply and another process waited on a timer. If the timer expired before the reply was received the timer process would resend the packet. Thus, the restriction of waiting on one message almost worked. Only in one case did it fail, but this one case was a total failure. The particular case is cleaning up and shutting down the process, especially after a protocol failure. In this case, it is necessary to get every process, presumably waiting for some specific message, to stop waiting, free its storage, and destroy itself. Creating a special process for the shutdown message does not solve the problem, for the shutdown message is of immediate interest to every process.

The handling of exceptional events has always been an area where the proper semantics are unclear. It is an actual overhead and a programming complexity to test constantly for all possible exceptions. Various mechanisms have thus been created to signal a process when an exception occurs, by diverting it from its normal execution path. This diversion has the problem that the signal, occuring at an unexpected time in the program, can find the process unprepared to process it.

It is not surprising that the problem of exceptions, which has always arisen as part of subroutine calls, should also arise as part of message sending. While it is not surprising, it is certainly regretable, for the suitable mechanism has never been clear even in that simpler case. The general problem is complex enough to demand a separate discussion below, but the first observation is that if a process can wait for only one message, then there is no way, clean or dirty, to inform it of exceptions, and this is an unacceptable restriction. Thus, a process must be able to wait for more than one message. At the same time, things can be arranged so that there are only a small number of messages expected under normal circumstances. Thus, a complex system mechanism to sort and filter incoming messages is not important. Thus the claim that waiting for any message is the proper system semantics.

Note that send-reply semantics cause problems with exceptions, because a process waiting for a reply is a specific example of waiting for one particular message. This is an additional defect in send-reply semantics.

3. Scheduling Implications of Message Passing

A most important issue related to message passing is the manner in which receiving a message causes a process scheduling. In general, of course, the required function is clear. If a process is waiting for a message, and a message arrives, then the process should be run to receive it. This statement is sufficient if the issue of performance is ignored. If, however, the system is expected to run well, not just run, the situation becomes much more complicated. This aspect of performance was particularly troublesome for the network protocol software since scheduling was such a large part of the total overhead in the first place.

The problem is that not all messages imply the need for immediate processing. Processing later, in an idle moment, is sufficient. One approach to this is to try to identify low priority processes, and have the system take account of this priority ordering. This is not at all sufficient for the network protocol project. What is required is a priority related to the message, not the process. There must be high priority messages, causing the receiving process to run at once (perhaps even pre-empting the sending process), and there are lower priority messages, for processing later as time permits.

Even this is not sufficient, for the sender and the receiver may disagree as to the priority of a message. A sender may attach highest urgency to a message, but if the receiver is otherwise prevented from proceeding, there is no reason to schedule it when the high priority message arrives. Thus, some sort of negotiated priority is required with every message.

At a minimum it must be possible to send a message and avoid causing any scheduling action as a result. Often, one process wants to send a number of items to another. For example, one process may parse an incoming data packet and find a number of lines of data within. If the interface to the client process is in terms of lines, then one message will be sent per line. Clearly, if performance counts, the client should not be scheduled until all the processing of the packet is done. If this simple control of scheduling is not possible, the code cannot be expected to run well.

In the protocol project, for any given message, a particular scheduling action that satisfies both sender and receiver could be found. The system does not need an interface that resolves desires that are actually conflicting. What is needed is a way for a waiting process, while it is not running, to leave behind enough information about its state so that a sender can figure out what should happen. It is hard to design system semantics to support this in general. An alternative structure, which shares the negotiating function between system and application, is discussed below.

4. Process Storage Allocation

Operating systems can be divided into those with virtual memory, and those without. For simple systems with only one address space, a number of problems can be avoided. However, if different processes have different address spaces, message passing gets more complex. First, as discussed above, both the sender and the receiver must be able to create and delete the message. More basic than that, both must be able to refer to the message to read and write its contents. Second, the message must be able to refer to associated information (such as a buffer of data) in a way that is valid in both address spaces.

There are two common solutions to this problem. First, the data can be made part of the message itself. In the case of network protocols, this causes hopeless overhead, because the data must be copied twice as a part of message sending, once into the packet and once out again. Actual experience makes quite clear that this sort of overhead cannot be tolerated. The other approach is to put the data in a portion of the address space common to both processes. The only option usually available is the kernel address space itself. This has the problem that the kernel space, being shared, must be allocated with care, and must be protected from being overwritten by defective programs. This control is usually achieved at the cost of kernel intervention on the data access, which can have the overhead of a extra copy operation.

In the protocol case, there is a possible way out, because the patterns of sharing are not general. Which processes are going to pass messages to each other is known in advance, and is determined by the specific function of the processes. Thus, it will be sufficient if two processes, knowing that they wish to communicate with each other, ask the kernel to create a common data area which is specific to that pair (or larger group) of processes. Within this area, buffers can be allocated without a great deal of system concern, because a failure (such as exhausting the store) will only be visible to the particular processes in question.

If such a shared storage area is created between processes wishing to communicate, that area can be used for the allocation of messages themselves. This solves the system level problem of controlling use of a system resource. The hesitation in doing this sort of allocation is that it prevents completely general message passing. Two processes with no shared area cannot send messages. There is no strong body of experience to suggest that general unstructured message passing is important, and some to suggest it is not. A small system area could be created to allow for any special cases. This point will receive more consideration below.

The excessive overhead of data copying as a part of message passing suggests that some sharing of address space must occur between co-operating processes. In fact, such storage is required for other reasons. As discussed above, sender and receiver sometimes need to share state to determine whether one should schedule another. Since messages cannot be used for this, the only obvious alternative is shared storage. It would be nice if message passing could somehow be the only means of communication between processes. Shared memory is very unstructured, and unstructured communication between processes is known to cause program bugs. Sadly, there seems no way out. The difference in overhead between process scheduling (however well tuned) and a reference to shared variables will always be enough to matter.

5. Cooperating Processes -- The Multi-Process Module

The traditional view of processes is that each process constitutes a separate domain, with only restricted modes of interaction across process boundaries. This note has assumed a rather different view, in which processes know quite a bit about each other. They reveal to each other, not only a shared data space, but information needed to control scheduling. In fact, this programming project suggests a rather different view of modularity in general, which helps to solve many of the traditional problems of cooperating processes.

In general, a collection of programs is divided into units which have a well defined function and interface. The term "module" is sometimes used to describe this sort of unit. (Other terms are "package" and "layer".) In a companion paper, "An Alternative Protocol Structure", it is argued at length that the traditional ideas of module and modularity are not sufficient for protocols. Briefly, the argument is that a protocol cannot be modularized using processes to define the module boundaries, because that causes unacceptable cost and loss of flexibility in crossing the boundaries. Nor can a collection of subroutines be viewed as a module, because of the specific need which most protocol modules have to execute asynchronously from their client modules. What is needed is a hybrid between a subroutine and a process. What was produced in practice was a module that consisted of one or more processes together with some subroutines that could be called from processes belonging to other modules.

The resulting module boundary did not match any obvious system provided entity, which caused some implementation problems. The structure was vey easy to deal with from a conceptual point of view, however. First, since the module contained several internal processes, it was possible to arrange for a separate process to wait for distinct messages. As discussed above, this makes easier the internal dispatching of arriving messages. Second, clients in external processes had a low overhead way to interact with the module, via the subroutines which they were permitted to call in their processes. These external subroutines provide the module interface that the clients can use, while the process structure is hidden.

Shared memory was used both between modules and internal to each module. Internal to each module, the memory was used as needed, to provide common state information between the internal processes and the external subroutines. Between modules, shared memory was used only in constrained ways, as a way of passing data buffers with high efficiency. Other constraints were imposed on the interaction between modules. In particular, one module never scheduled the internal process of another process. The only way one module could activate another was by calling one of the available subroutines. All process scheduling actions occured internal to one module, even if parts of the module were running as subroutines in processes of other modules.

This multi-process module idea could be used as the basic structuring tool of a system. Several substantial advantages seem to result from doing so. In particular, several of the problems discussed above can be solved if all modules can be assumed to have an external part that runs in foreign processes.

The problem of linking message passing to scheduling is solved if one module never schedules another module, but only calls its external subroutines. As was observed above, experience with protocols suggested that in any specific case two processes can agree on the priority of any scheduling action, provided that enough information can be made available. The problem with this idea is making that information available across a module boundary. When one process calls an external subroutine of a module in order to have part of that module scheduled, all scheduling decisions are pushed inside one module, where it is easy for the various parts to interact however needed to provide the correct decision.

The problem of exceptional conditions is also solved by this structure, but it is necessary to consider exceptions in more detail in order to see the real advantage. The usual approach to exceptions (assuming that they are handled in any systematic way at all) is to provide some set of exception identifiers, either integer codes or character strings, which can be returned to signal the exception. In the context of subroutine calls, these can either be passed as an extra return argument, which the program must explicitly test, or as a software interrupt, for which the program must have provided a defined handler. Extending this idea to message passing, either form can be added to the system interface used to wait for a message: either a software interrupt can occur at this point or a return with error code can occur. All of the programming problems associated with either approach still occur, but no worse than in the subroutine case.

The real problem with exceptions is that an exception can occur, not when a program is itself running, but when it has called some other program on its behalf. Returning to the familiar case of subroutines again, an event of interest to one subroutine can occur when it has called another, which is currently executing on the stack. The exception usually means nothing to the subroutine, but it is expected to "do the right thing" anyway, perhaps to clean up in some way and clear itself off the stack. The real problem with exceptions is to get the other modules involved to "do the right thing".

With message passing, the real problem occurs when, inside a process, some subroutine is called and that subroutine waits for a message. For example, the system routine to read from the keyboard is called, and that waits for a message from the terminal handler. If an exceptional event occurs while that subroutine is waiting, it may be necessary to abort the wait, even though the subroutine knows nothing about the exception in question.

The multi-process module provides an elegant way to solve this problem. When one process of a module detects an exception of interest, it causes that event to be noticed in other processes of the module by explicit action. If a process is currently in the external subroutine of some other module, waiting for a message, this process will call another external subroutine of that other module, provided for the purpose, instructing it to abort the wait.

It may not be obvious that simplicity is best served by forcing every multi-process module to provide various external subroutines just to abort waits and so on to help with exceptions in other modules. In fact, this is a very simple and clean idea. Most modules have a very limited number of exception responses. It is not a burden to design and implement them. What is a burden is for a module to try and guess which is appropriate to perform in response to an exception which is defined by and meaningful to some other module. In this scheme, using the external subroutine interface to the module, the module can be explicitly requested, using terms meaningful to the module itself, to perform whatever action is appropriate.

A final area where the multi-process module seems to help is in resolving the naming of message types. When a process receives a message, it must be able to tell which message arrived, so that it can proceed accordingly. In some systems, every message has a unique id, but this is not often helpful, because the real issue is what "type" of message is it. Some systems, such as Unix, distinguish messages based on the sending process. This is quite restrictive. What is needed is for each message to carry some "type identifier" which the receiver can examine.

If messages are sent in an unconstrained manner between all pairs of processes, however, then the type identifiers must be globally assigned, which creates fully as complex a naming structure as global file names. Especially since a compact representation such as integers is needed for message type identifiers, the management problem is severe. But if messages are sent only between processes of one module, then the scope of type identifiers is restricted to that module, which is tractable.

6. System Support for Multi-Process Modules

As discussed above, the multi-process module was an abstraction that was not directly supported by the system. What the system supported was the various process abstractions, message passing, scheduling, etc. out of which the module was built. However, if the multi-process module were officially supported by the system, a number of problems would be solved.

Most important, the system could supply a class of storage suitable for communication between the processes of the module. In most systems today, the idea of the external subroutine stresses the storage problem hardest, because the external subroutine, running in the process of some other module, must be able to find its shared storage. One way to look at this is that the external subroutine links two address spaces together, those of the calling and the called modules. In fact, the idea of address space is totally altered by the multi-process module structure. There are now two kinds of shared storage, the state information internal to a module, and the communication area linking two modules, which is used for shared data between them. The address space for any particular process is thus a stack (or similar mechanism) plus the internal and external storage and code segments of all the modules that may be run on that stack. Each external subroutine must be able, by knowing its module membership, to find its storage in the address space of the calling process. This mapping would be made much easier if the system understood the idea of module and maintained the proper sort of mapping tables.

In fact, there are actually two forms of multi-process module. The module "template" defines the needed internal processes, the external subroutines, and the storage requirements. The module "instance" is one particular version of the module, defined by one version of the storage. When one module calls another, what is invoked is a particular instance. The instance name is often one of the parameters of the call, so that the called routine can figure out what is being called. For example, for a protocol layer there is usually one instance per connection, and the layer is invoked with the connection name as one argument of the call. Thus, the external subroutine of a module must be able to find the correct address space based on which instance is currently being called. Instance names must thus be a system concept.

Normally, this idea of "instance" is found in the process abstraction: the same program running in different processes is a different version with different state. In this structure, where the module is the system entity with multiple instances, the idea of process, although still present, is much less important. A process has only a stack to distinguish itself, except when it is waiting, in which case it has certain messages in which it is interested. If every module is viewed as existing in versions, and all versions are distinguished by storage, then even the stack is not needed, except to keep the actual sequence of return addresses. Stack or automatic storage is replaced by version storage. This idea could be helpful on machines in which the hardware support of multiple stacks is somewhat weak, or the number of address registers limited.

The programming structure of the module is another area in which system support might prove useful. Experience coding in this style suggests that the programmer will not use the internal processes and external subroutines of a module in arbitrary fashion. In fact, one of two patterns emerged. In one, there were no internal processes at all, because the module had no requirement for asynchronous actions. Note that in this case, the module is still multi-process, for it could be called from any of the processes of the client module. In the other pattern, all of the work was done in the internal processes, and the external subroutines only did interface tasks, such as setting state variables, adding messages to queues, and scheduling internal processes. In particular, the external subroutines never waited for messages. This last pattern is a rather clean one, in that it is easy to specify the behavior of the external subroutine interface. System support of this structure might prove helpful.

7. Conclusion

This paper has proposed an operating system with some rather unusual features. While it is based on message passing, it requires that processes share memory, and that scheduling not be tied too tightly to messages. It has proposed a new abstraction for program modularity, called the multi-process module. This module provided a restricted domain within which message passing was used, which solved various otherwise difficult system problems, in particular efficient scheduling, multi-process exception handling, and storage management for shared memory and messages. The storage requirements of the multi-process module in turn implied a rather different view of system storage management, which separated the ideas of address space and process, and associated storage with module instances rather than processes.

In fact, the message seems a rather unimportant aspect of the final design. It could be argued that the real conclusion of this paper is that message passing semantics are not very useful. Such a conclusion might be defended, but seems to overstate the case. In fact, message passing plays an important part in this structure.

First, messages can still be the vehicle for scheduling. Just because it must be possible to send messages without triggering scheduling, it does not follow that the two are unrelated. One idea that is consistent with these proposals is to have scheduling controlled by a parameter to the send message routine.

Message queues must be system objects. This is so, first, because it is on a message queue (or port, or whatever name appeals) that a process waits, clearly a system operation. The system must name queues, because those names are global, at least with respect to any one process. Equally important, the actions of queueing and dequeueing a message must be system atomic actions. This is so because a sending and a waiting process must be sure that they have avoided any race conditions in testing an empty queue.

Since queue management is atomic, messages can be used to build a synchronism mechanism between co-operating processes. This point has not been considered before, but it must be clear to the careful reader that programming a multi-process module, since it involves parallelism, requires care and co-ordination. Some form of interlock will clearly be needed, and the message can provide this function. Since messages can cause scheduling, this makes it possible to wait on a lock by waiting for a message. Essentially, a message is a lock. While dequeueing a message may not seem as efficient as a test and set instruction, the overhead of building a test and set if the hardware does not have it is about the same as the operation of dequeueing, which is in detail a test and set sort of action.

The argument of this paper, then, is not that messages are useless, but that they must be used in structured and constrained ways, as must locks, goto, memory management, and many other tools, if they can be implemented by the system and understood by the user.