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

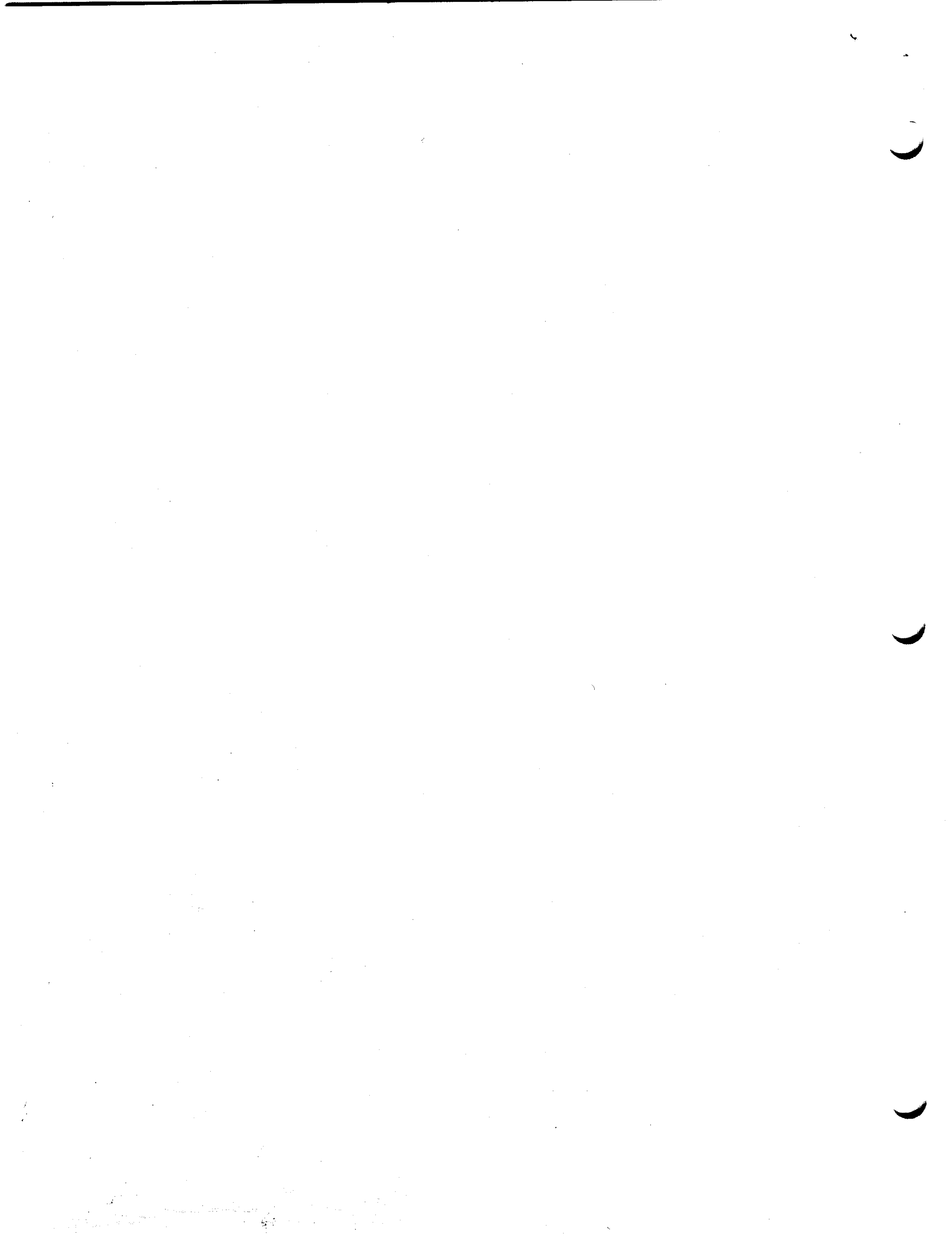
Computer Systems Research Request for Comments No. 223

An Alternative Protocol Implementation

by David D. Clark

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

WORKING PAPER -- Please do not reproduce without the author's permission, and do not cite in other publications.



COMPUTER LABORATORY, CAMBRIDGE

Systems Research Group Note

David D. Clark - 10 May 1982

An Alternative Protocol Implementation

1. Introduction

There is a serious design problem that arises in the implementation of a network protocol on an operating system, which is how to employ the facilities of the operating system in structuring the implementation. There are two facilities normally available in an operating system, the procedure and the process. Unfortunately, protocols, because of their particular requirements, do not map well onto either of these mechanisms. The purpose of this note is to discuss why this is so, and to describe an experimental implementation that attempts to explore a new structuring technique well suited for protocols.

Protocols are always specified as being layered, which tends to suggest to the implementor that each protocol layer should be a distinct module of some sort in the implementation. Unfortunately, the two obvious module types, the procedure and the process, both fall short. If a layer is realized as a procedure, then the client of that layer views the layer as a subroutine package to be called as needed in the process of the client. This provides a very efficient interface between the client and the layer, since a subroutine call is usually inexpensive, but it does not provide any means by which the layer can run except when called by the client. Unfortunately, most protocols contain the concept of an asynchronous action, something the protocol layer performs as part of its own operation and not at the request of the client. For example, a unit of data transmitted may produce a returning acknowledgment. The client is not interested in this acknowledgement, but the protocol must run at the time it arrives to process it. This need to run asynchronously means that the abstraction provided by the routine is not sufficient for most protocol layers.

The alternative is implementing the layer as a separate process. This clearly provides the necessary asynchronous execution, but now fails to provide an interface to the client that is efficient or flexible. Process scheduling, which is now required to invoke the layer on behalf of the client, is usually more costly by far than subroutine invocation. Moreover, interprocess communication is usually implemented with sufficient limitations that it does not really support the needs of the

layer interface. On some systems that do not share memory between processes, a layer invocation may even require a copy of data through the kernel of the system, which can cause unworkable overhead. The difficulty of dealing with this interprocess interface contributes to a rather surprising result, which has been repeatedly observed in protocol implementations: the amount of code required to deal with the foreign machine is found to be much smaller than that required to deal with the client layer above.

There is a third way in which a protocol can be realized; it can be made a part of the operating system kernel itself. Inside the kernel, things usually get somewhat less structured, so it is possible to provide a special mechanism to support the protocol layer. However, some systems have severe restrictions on the address space of the kernel, so that the layer and its data simply will not fit. More importantly, the facilities available in the kernel are often a subset of what is available to the user; for example, there may not be the concept of processes within the kernel, so that the only mechanism available for asynchronous actions is the interrupt handler. Thus, kernel implementations are tricky to build and hard to maintain or change. Finally, when the trend in operating system design is to try for a very small kernel with almost all the function moved elsewhere, the idea of a very large protocol package moving into the kernel is very hard to accept.

2. Modularity vs. Efficiency

Part of the justification for the design presented in this paper is the claim, to be discussed below, that traditional layering techniques produce a protocol modularity that is unsuitable for high performance. An examination of the performance constraints on protocols will show that they are very severe. Thus, it is worth taking a moment to consider the issue of efficiency, before considering how it conflicts with modularity and layering.

There are many aspects to efficiency. One aspect is sending data at minimum transmission cost, which is a critical aspect of common carrier communication, if not in local area network communication. Another is sending data at a high rate, which may not require special effort if the net itself is very slow, but which may be the one central design constraint when using a net with high raw bandwidth. The final consideration is doing the above with minimum expenditure of computer resources. This last may be necessary to achieving high speed, but in the case of a slow net may be important only in that the resources used up, for example CPU cycles, are costly or otherwise needed. It is worth pointing out that these different goals often conflict; for example it is often possible to trade off efficient use of the computer against efficient use of the network. Thus, there may be no such thing as a successfull general purpose protocol, either in implementation or design.

There are two general rules that help to achieve any of the above goals. First, send the data using the minimum number of packets, and second, process each packet with the minimum overhead. Clearly, one must process with low overhead. A simple computation will show that the restrictions are very severe. Consider the following table, which shows the maximum average interval between data packets which must hold if one is to achive a particular throughput with packets of various sizes.

		Packet size in bits.	
		1K	10K
Throughput	10KBPS	100ms	1sec
	100KBPS	10ms	100ms
	1MBPS	1ms	10ms
	10MBPS	.1ms	1ms

The point of this very simple table is that if one wants, for example, to send data at an actual rate of 1 megabit per second, and each packet has 10,000 bits in it, which is about as large as packets get in practice, each data bearing packet must follow the last within 10ms. This 10 ms includes not only the actual data transmission time but the system processing time. The sending host must prepare the data and format the packet, and the receiving host must do the reverse. In some protocols, there may also be the overhead of an acknowledgement packet which must flow back from receiver to sender. If we assume that the sender and receiver have equal processing costs, and we assume a 10MB raw data speed, then 1ms of the 10 available is used in transmission, and the rest is divided equally, giving 4.5ms at each end. Of course, cunning design can allow both machines to execute at the same time, but even then there is at most 10ms, which is a very hard limit to meet in practice.

One does not usually find 10ms of protocol related processing per packet. In fact, the important cost often turns out to be system overhead, and not protocol related costs. There exist systems which take most of 10ms just to do a process scheduling, and protocols implemented as processes may require many process schedulings. For a variety of protocols and systems, examination of actual implementations suggest that throughput is limited by the system cost dictated by the modularization of the protocol, rather than by the details of the protocol processing at any one layer.

This last point makes clear why efficiency and modularity relate to each other. Aside from the obvious point that good throughput requires large packets, all the issues above are influenced by modularity considerations.

First, a protocol should be structured, in implementation if not in design, to reduce the number of system operations (such as process swappings) that occur on each transaction. No protocol, to my knowledge, has taken this into account in the design of its layering, so it follows that one cannot expect the layering boundaries to be obvious in the realization.

Second, protocols should be structured to minimize asynchrony. This point is not as obvious, but is critical. The term asynchronous operation was used above with respect to a layer to describe an action that layer takes as part of its own operation, independent of the immediate desire of the client layer above. The example was processing an acknowledgement. In traditional protocols, most layers have some functions that require this sort of action. The problem is that a packet sent as part of an asynchronous action, precisely because it is asynchronous, does not have client level data in it.

Of course, one can hope that an outgoing packet can be shared by several layers, assuming that several layers happen to have an asynchronous action to perform at the same time. The most common expression of this hope is found in protocols where an acknowledgement is capable of being carried on a reverse direction data packet. In fact, it is hard to find an implementation in which this reverse direction "piggy-back" is actually achieved. The problem in making this work is precisely that the layer boundary creates a restricted interface across which the lower layer is incapable of asking the client if there is any use that the client can make of an asynchronous packet which the lower layer is sending out. In practice, then, sharing of a packet by different layers does not occur. Each layer sends its own packets, which contributes to processor overhead and to packet related communications costs.

The last problem of modularity relates to the observation above that it is often harder in practice to provide a suitable client interface than to provide the protocol implementation to talk to the foreign machine. The reason for this seems to be that a layer is viewed as supporting many different clients with different performance requirements, which implies that the layer must be general enough to meet all of these different performance requirements in one realization. For example, a reliable stream package may be called on to support remote login, where the prime consideration is low delay for packets with one byte of data, and at the same time to support file transfer, which requires efficient handleing of the data in maximum size packets. To meet goals of this diversity requires very sophisticated buffering and coordination algorithms, which are more bulky and complex than the protocol itself.

3. An Alternative Approach

The preceding discussion provides the justification for a protocol implementation with a rather novel architecture. The design goals were the following. First, the implementation should provide a structure which permits necessary asynchrony while providing an efficient interface between the several layers. Second, it should permit several clients to share a common layer without complex programming in that layer. Third, it should permit sharing the overhead of asynchronous actions by different layers.

The structure devised for the implementation has two important aspects, the use of processes and the client interface. In this realization, a single layer is not represented a either a set of subroutines or as a process, but as both together. The client is provided with a set of subroutines which it may call in its process. Along with these routines are some number of separate processes, which perform any asynchronous actions. All communication between processes is internal to a single layer; layer crossings are done via subroutine calls. For some layers, all of the work can be done in the subroutines, and no processes are used.

The client interface is the other odd aspect of this structure. Traditionally, one thinks of a client invoking a lower layer by calling it. That is, calls proceed downward through the layers. This, however, is not always the right direction. It is often better for the call to come from the lower level to the client. This is the obvious structure when a packet arrives. The packet must first be processed by the lowest level, and the higher level must be called into play only after valid higher level data has been determined to be in the packet. An up call to the client is the obvious way to obtain this structure. It might be thought that on sending a packet the structure would be upside down from receiving it, but this is not so. A packet to be sent is built from the front back, or (in terms of layering) from the bottom up. That is, the first part of the header is that belonging to the lowest layer. An up call is thus quite natural for building a packet to send. The bottom layer fills in its header first, and after doing so, calls up to the client to fill in its part at the proper offset. The only ways to do this if down calls are used are either to let the lower layer copy the client data into the packet, which is a useless copy needed only for reasons of bad layering, or to build in to the client the knowledge of the lower layer header size, which is clearly a bad layer violation and may not even work if the lower level header can be of variable size. Thus, up calls rather than down calls turn out to be a much more natural structure.

In fact, up calls have a second advantage, perhaps as important as the natural fitting to function claimed above. The up call structure makes it much easier for a layer to ask advice of its client as it is needed. The problem here is that in traditional down call structures, the client asks the lower layer to do something, but that action may not occur at once. For example, the client may wish to have some data sent, but the lower layer may not be able to send it just then because of a flow control restriction. So the data is sent some time later as an asynchronous action. By the time the data is sent, the client requirements may have changed. For example, the client may have additional data by now. It would be more straightforward if the lower layer were to call up to the client only when the packet were actually being sent to ask what use the client could make of it. This means that up calls should be allowed to occur when the lower level needs them, not when the client level wants them. In other words, up calls are asynchronous with respect to the client.

The clearest example of the advantage of asynchronous up calls occurs when an incoming packet arrives. The lower level, on determining that a packet contains valid client data, must get that data out of the packet and into the client. In traditional structures, the client is not assumed to be on call at the moment a packet arrives, so the lower layer must put the data into a buffer until the client can be notified to get it. This buffering algorithm used by the lower layer must be very general, for it must be suitable for whatever sort of client wishes to use it. It is for that reason that the interface between a layer and its client is traditionally so complex. In contrast, the up call approach is extreamly simple. At the time that the lower layer wants to remove the data from the packet, it simply calls up to the client and instructs the client to remove the data now. Of course, this does not remove all need for buffering. What it does do is move the buffering function from the lower layer to the client. This structure is much better. The lower layer, in this approach, will be found to consist of only the actual protocol processing parts, those related to dealing with the foreign machine. This makes most protocol packages get much smaller and easier to understand. The client can implement a buffering strategy that is tailored to its needs, which can be expected to be much simpler than the general mechanism the lower layer would have otherwise provided. Thus the client does not grow nearly as much as the lower level has shrunk. This general reduction in bulk and complexity does seem to occur in practice.

There is one further aspect to this structure, which is that it provides a way to promote the sharing of packets between several layers. A specific example will make this clear. When an incoming packet contains data, a stream layer will want to acknowledge that data. It may also be the case that a higher level will want to send data in return. When providing remote login service with remote echo, for example, the stream layer will want to acknowledge each character arriving, and the next level up will want to echo each character arriving. To reduce overhead, it is quite important to get both of these returing items in one packet. As discussed above, in traditional structures, this is very hard to do, since the stream layer, knowing nothing about the client layer, does not know that an echo is an almost inevitable response to an incoming character. It has no way of asking. Either it must wait and see if the client provides a response, which injects needless delays if the goal in this particular case was throughput, or it can just ignore this sharing and send an acknowledgement on its own. In the up call structure, this problem can be approached in several ways. The simplest is for the stream layer to call up to ask whether data will be soon available. Even more direct is to have some of the client code run as part of the up call to compute the echo value right then. In any event, it is easy for the lower layer, by using up calls, to tailor itself to different client modules without building hugh general purpose interface packages.

4. An Example: BSP

An experiment was carried out on this protocol structure using the Tripos operating system, and the family of protocols developed by the University of Cambridge: BSP, a byte stram protocol, and VTP, a protocol supporting remote logon. A brief description of this implementation, BSP in particular, will help make the above discussion more concrete.

Tripos has two mechanisms for producing asynchronous or parallel actions, the task and the coroutine. The task is a system level process with a system specified address space and scheduling mechanism. Coroutines are a language level multiplexing of one task. Coroutines are very inexpensive to schedule, and it was possible to build a per task coroutine scheduler that implemented the particular features required. For these and other reasons related to details of Tripos, one task was created for each network connection, with its own BSP, VTP, etc. running inside it. Internal to each protocol in the task, coroutines were used tp provide the asycnchronous actions.

The coroutine scheduler, which provids the operating environment to tie together the parts of each layer, is sufficiently important to deserve attention itself. The scheduler is not based on the idea of static priority, but on the idea that, at the time one coroutine activates another, the activating coroutine will specify how the other task is to run. Four activation modes were supplied:

Pre-empt: Suspend the activating task and run the other task. When that task finishes, come back and run this task again. This form of activation is somewhat like a subroutine call.

No Delay: Add the activated coroutine to the list that the scheduler has of runnable coroutines. The scheduler will examine that list and select another coroutine to run when this activating coroutine finishes and suspends itself.

Short Delay: Mark the relevant task as runnable after a short fixed period (currently 100ms.) has passed. This short period is used to postpone sending an acknowledgement to see if the client layer will generate data to combine with it. It has other similar uses.

Long Delay: Mark the relevant task as runnable after a longer period (currently 5 seconds). This interval is used to control retransmission.

As the above suggests, the important idea in the scheduler is not priority, but timers. In fact, much of what the scheduler does is setting and clearing timers, which suggests that any system supporting this sort of software should have a good package for timer management. In fact, most timers never go off, for they are only to protect against some failure. Thus, the efficient operations on timers must be setting and clearing, and not responding when they go off.

The BSP module consists of two coroutines and a set of subroutines to be called from the coroutines of other protocol layers. One coroutine waits for arriving packets from the net, the other sends packets to the net. As discussed above the sending activity as well as the receiving

action is properly thought of as an asynchronous action, since a packet cannot always be sent whenever the client desires.

There are several subroutines that the client (such as VTP) can call in BSP. These are structured to do no actual work, but only to schedule coroutines and set flags as necessary before returing. The calls are as follows:

Send: Request BSP to run its send coroutine to transmit some data. BSP will do this as soon as flow control and foreign acknowledgement permit.

Receive: Request BSP to undo a previously requested flow control halt to incoming data, so that more data may come. (In BSP terms, this causes the BSP Ready command to be sent. This is only required if a NotReady was previously sent.)

Send Reset: Requests BSP to resynchronise the stream. BSP will do this at once, using the send coroutine to do so.

Send Close: Requests BSP to start the exchange to close the connection. The send coroutine will attend to it.

Abort: Request the BSP to free storage and finish, on the assumption that the other side is dead. A drastic form of close.

Open: Request BSP to play the active role in starting a connection. In this case the coroutine actually calling the subroutine is used to execute the sending and waiting associated with opening. In this protocol family, the opening sequence is not part of BSP, but is a separate layer. It would thus unnatural to use the BSP coroutines for this. The higher level client must expect that its coroutine may be detained for some time inside the Open call. It will return only when the open has succeeded or failed.

Listen: Request BSP to play the passive role in starting a connection. As above, the action is performed using the coroutine of the calling client.

An examination of the above calls will suggest that not much actually happens there. In fact, all of the interesting function is performed as part of the up calls from BSP to the client. Those calls are as flollows:

TakeData(ptr,count): The client is given a pointer to count bytes which are contained in a packet just arrived. The client is expected to (quickly) remove the bytes from the packet.

GiveData(ptr,count): The client is given a pointer to an area in a packet about to be sent. It is expected to put some data there and indicate the actual number of bytes given.

TakeFlags(flags): The client is given the control flags that are transmitted with every BSP data packet. It is assumed that it will in turn request the proper action. In particular, if the Close Request flag is on, the client will probably want to call the Send Close subroutine of BSP.

GiveFlags: The client is requested to supply the control flags to be included in the packet now being sent.

ResetComplete(who): Indicates to the client that a resynchronisation has occured. The argument indicates who asked that this happen (It could have been the client itself, via the Send Reset down call.

CloseComplete(who): Indicates to the client that the connection has been broken. The arguments indicates which end initiated this close.

TakeP: This call is a query of the client to determine if it can take more data. If the answer is no, BSP will halt the foreign host (by sending Not Ready instead of Ready as an acknowledgement of the packet currently being processed). In this case, the client must call Receive at the time when data can again be received.
GiveP: This call is a query of the client to determine if it has more data to send. If so, the BSP send coroutine will be started to send that data.
HowFastAckP: This call is a query of the client to determine how fast the acknowledgement of the last data should be returned. If the client desires maximum throughput it will request immediate acknowledge. If the client wishes to combine that acknowledgement with some outgoing data, as often happens with remote login, the client will request a Short Delay be used to schedule the send coroutine. (When the data is ready, the client can use the Send Data call to override the Short Delay.)
Error(how many): This call imforms the client that "how many" errors have occured since the foreign host did anything right. Errors include timeout and retransmission, which is the normal error, but also include malformed packets received and similar misbehavior. This call will not be made until "how many" reaches a set minimum. The client, unless it has special knowledge that justifies these errors, should respond by calling Abort.

These up calls can be grouped together according to when they are called. TakeData,TakeFlags,TakeP,GiveP and HowFastAckP are called as part of processing an incoming packet and deciding what sort of response to send. ResetComplete and CloseComplete are called as part of processing an incoming control packet. GiveData and GiveFlags are called as part of actually sending a packet. Error may be called at any time. This sort of information, which describes the timing or sequencing, is an important part of the specification, since it allows the client to know what to expect. The problems associated with sequencing are discussed below.

Inside BSP, the two coroutines and the subroutines communicate among themselves by a shared area of storage, which also constitutes the state information of BSP. It takes 22 integer variables to hold this state, which seems reasonable. The particular operating system used, however, does not provide any effective way to create a class of storage suitable for this. The only option is to use the global storage that is allocated on a per task basis, which means that it is impossible to put more than one BSP in a task, to avoid conflict over global storage allocation. This is not a problem particular to Tripos, most operating systems do not have exactly what is needed.

5. Problems

The structure described above has several clear advantages. The various layers do not require much code, and they interact with each other in a flexible and efficient manner. There are as well several problems with this structure. Many of these problems do not relate to

the limitations of this particular implementation, but rather to general practices in the design of protocols and operating systems. It is thus worth a careful study of these problems.

The multi-process structure used above is very different from traditional practices in protocol modularization. Thus, packages implemented in this new way to not interface well to packages done the old way. This is not surprising, but it must be remembered when contemplating the size of a redesign project. More importantly, this structure, where one layer lives in several coroutines at once, some of which are controlled by the layer itself and some of which are controlled by other layers, is a complex structure which is hard to understand and debug. One way of expressing the problem is that some of the protocol structure is captured in the program, not as a sequence of statements on a page, but as the sequence of scheduling events that occur. A scheduling event causes some other piece of code to come into execution at some unknon time. To understand the impact of this style of coding on the programmer, on can usefully consider the scheduling event as similar to an unstructured GOTO with only a probability of executing. Of course, other styles of protocol implementation have this same problem.

The problem of linking the various coroutines of a layer together by means of shared variables was discussed above. This structure would not work if the only means of communication between coroutines was by message passing. Not only would that cause unworkable overhead, but it would make the structure iompossible. If one puts part of the state of a layer in a message and sends it to a coroutine, that part of the state has become inaccessible for the time while it sits on the input queue of that coroutine. The other coroutines must thus be prepared to do without that part of the state information until that coroutine has run. But that sort of constraint is inconsistant with the unpredictable order in which things can happen when one is talking to the outside world. It would be nice to think that message passing would eliminate the need for shared memory, which is so unstructured. That goal does not seem possible.

Dispite the above, the message passing facilities of the system need to be very powerful to support this structure properly. A companion paper discusses what has been learned about message passing from this project.

Each coroutine, although it belongs to one layer, makes calls out of that layer into other layers as part of its normal operation. A layer thus called may in turn call other layers, so the amount of stack needed for each coroutine is difficult to predict. Unfortunately, most coroutine or task packages require the maximum size of each stack to be known at creation time. The only way out is to create big stacks, which is hard on small memory systems.

Perhaps the most difficult aspect of creating this structure is that, because of the up calls, a layer gives up control to another layer in the middle of its execution. This was, in fact, the goal, because it yielded a very flexible and efficient interface. The difficulty is that

one layer can never be quite sure what another layer is going to do while it has been called. For example, when BSP calles its client to take some data, the client may, while it is running in the coroutine of BSP, turn around and call back down into BSP to request some other action, perhaps to send a reset or a close. Having done this, the client will then return to BSP, which will discover that all its state has unexpectedly charged. It is difficult to create a program that knows how to go on under these conditions.

There are several ways to solve the problem of the side effects of up calls to the client. One is to prohibit, as a part of the layer specification, any execution of a down call in that piece of client code that executes on an up call. This, when tried, caused great complexity in the client layer. There are several side effects that are really needed as part of normal operation. The layer specification, instead of prohibiting side effects totally, could list those that are acceptable. The attempt to do this for BSP led to a very complex specification, which seems wrong in spirit and dangerous in practice. The best solution seemed to be for each layer to protect itself from whatever the layer above could do.

BSP achieved this by making all the down calls do nothing but set a very restricted set of flags and schedule the internal coroutines. BSP is thus safe from side effects if it rechecks the relevant of these flags after each outcall. In fact the number of such tests required is not large, but the problem is hard to think about, and can easily lead to difficult, timing related bugs. It is regretable to create a structure in which this sort of disaster is a known feature. More experience wiht this style may provide better insight into dealing with up call side effects.