RFC: 817

MODULARITY AND EFFICIENCY IN PROTOCOL IMPLEMENTATION

David D. Clark
MIT Laboratory for Computer Science
Computer Systems and Communications Group
July, 1982

1. Introduction

Many protocol implementers have made the unpleasant discovery that their packages do not run quite as fast as they had hoped. The blame for this widely observed problem has been attributed to a variety of causes, ranging from details in the design of the protocol to the underlying structure of the host operating system. This RFC will discuss some of the commonly encountered reasons why protocol implementations seem to run slowly.

Experience suggests that one of the most important factors in determining the performance of an implementation is the manner in which that implementation is modularized and integrated into the host operating system. For this reason, it is useful to discuss the question of how an implementation is structured at the same time that we consider how it will perform. In fact, this RFC will argue that modularity is one of the chief villains in attempting to obtain good performance, so that the designer is faced with a delicate and inevitable tradeoff between good structure and good performance. Further, the single factor which most strongly determines how well this conflict can be resolved is not the protocol but the operating system.

2. Efficiency Considerations

There are many aspects to efficiency. One aspect is sending data at minimum transmission cost, which is a critical aspect of common carrier communications, if not in local area network communications. Another aspect is sending data at a high rate, which may not be possible at all if the net is very slow, but which may be the one central design constraint when taking advantage of a local net with high raw bandwidth. The final consideration is doing the above with minimum expenditure of computer resources. This last may be necessary to achieve high speed, but in the case of the 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 successful general purpose protocol implementation.

The simplest measure of performance is throughput, measured in bits per second. It is worth doing a few simple computations in order to get a feeling for the magnitude of the problems involved. Assume that data is being sent from one machine to another in packets of 576 bytes, the maximum generally acceptable internet packet size. Allowing for header overhead, this packet size permits 4288 bits in each packet. If a useful throughput of 10,000 bits per second is desired, then a data bearing packet must leave the sending host about every 430 milliseconds, a little over two per second. This is clearly not difficult to achieve. However, if one wishes to achieve 100 kilobits per second throughput,

the packet must leave the host every 43 milliseconds, and to achieve one megabit per second, which is not at all unreasonable on a high-speed local net, the packets must be spaced no more than 4.3 milliseconds.

These latter numbers are a slightly more alarming goal for which to set one's sights. Many operating systems take a substantial fraction of a millisecond just to service an interrupt. If the protocol has been structured as a process, it is necessary to go through a process scheduling before the protocol code can even begin to run. If any piece of a protocol package or its data must be fetched from disk, real time delays of between 30 to 100 milliseconds can be expected. If the protocol must compete for cpu resources with other processes of the system, it may be necessary to wait a scheduling quantum before the protocol can run. Many systems have a scheduling quantum of 100 milliseconds or more. Considering these sorts of numbers, it becomes immediately clear that the protocol must be fitted into the operating system in a thorough and effective manner if any like reasonable throughput is to be achieved.

There is one obvious conclusion immediately suggested by even this simple analysis. Except in very special circumstances, when many packets are being processed at once, the cost of processing a packet is dominated by factors, such as cpu scheduling, which are independent of the packet size. This suggests two general rules which any implementation ought to obey. First, send data in large packets. Obviously, if processing time per packet is a constant, then throughput will be directly proportional to the packet size. Second, never send an

unneeded packet. Unneeded packets use up just as many resources as a packet full of data, but perform no useful function. RFC 813, "Window and Acknowledgement Strategy in TCP", discusses one aspect of reducing the number of packets sent per useful data byte. This document will mention other attacks on the same problem.

The above analysis suggests that there are two main parts to the problem of achieving good protocol performance. The first has to do with how the protocol implementation is integrated into the host operating system. The second has to do with how the protocol package itself is organized internally. This document will consider each of these topics in turn.

3. The Protocol vs. the Operating System

There are normally three reasonable ways in which to add a protocol to an operating system. The protocol can be in a process that is provided by the operating system, or it can be part of the kernel of the operating system itself, or it can be put in a separate communications processor or front end machine. This decision is strongly influenced by details of hardware architecture and operating system design; each of these three approaches has its own advantages and disadvantages.

The "process" is the abstraction which most operating systems use to provide the execution environment for user programs. A very simple path for implementing a protocol is to obtain a process from the operating system and implement the protocol to run in it. Superficially, this approach has a number of advantages. Since

modifications to the kernel are not required, the job can be done by someone who is not an expert in the kernel structure. Since it is often impossible to find somebody who is experienced both in the structure of the operating system and the structure of the protocol, this path, from a management point of view, is often extremely appealing. Unfortunately, putting a protocol in a process has a number of disadvantages, related to both structure and performance. First, as was discussed above, process scheduling can be a significant source of real-time delay. There is not only the actual cost of going through the scheduler, but the problem that the operating system may not have the right sort of priority tools to bring the process into execution quickly whenever there is work to be done.

Structurally, the difficulty with putting a protocol in a process is that the protocol may be providing services, for example support of data streams, which are normally obtained by going to special kernel entry points. Depending on the generality of the operating system, it may be impossible to take a program which is accustomed to reading through a kernel entry point, and redirect it so it is reading the data from a process. The most extreme example of this problem occurs when implementing server telnet. In almost all systems, the device handler for the locally attached teletypes is located inside the kernel, and programs read and write from their teletype by making kernel calls. If server telnet is implemented in a process, it is then necessary to take the data streams provided by server telnet and somehow get them back down inside the kernel so that they mimic the interface provided by local teletypes. It is usually the case that special kernel

modification is necessary to achieve this structure, which somewhat defeats the benefit of having removed the protocol from the kernel in the first place.

Clearly, then, there are advantages to putting the protocol package in the kernel. Structurally, it is reasonable to view the network as a device, and device drivers are traditionally contained in the kernel. Presumably, the problems associated with process scheduling can be sidesteped, at least to a certain extent, by placing the code inside the kernel. And it is obviously easier to make the server telnet channels mimic the local teletype channels if they are both realized in the same level in the kernel.

However, implementation of protocols in the kernel has its own set of pitfalls. First, network protocols have a characteristic which is shared by almost no other device: they require rather complex actions to be performed as a result of a timeout. The problem with this requirement is that the kernel often has no facility by which a program can be brought into execution as a result of the timer event. What is really needed, of course, is a special sort of process inside the kernel. Most systems lack this mechanism. Failing that, the only execution mechanism available is to run at interrupt time.

There are substantial drawbacks to implementing a protocol to run at interrupt time. First, the actions performed may be somewhat complex and time consuming, compared to the maximum amount of time that the operating system is prepared to spend servicing an interrupt. Problems can arise if interrupts are masked for too long. This is particularly

bad when running as a result of a clock interrupt, which can imply that the clock interrupt is masked. Second, the environment provided by an interrupt handler is usually extremely primitive compared to the environment of a process. There are usually a variety of system facilities which are unavailable while running in an interrupt handler. The most important of these is the ability to suspend execution pending the arrival of some event or message. It is a cardinal rule of almost every known operating system that one must not invoke the scheduler while running in an interrupt handler. Thus, the programmer who is forced to implement all or part of his protocol package as an interrupt handler must be the best sort of expert in the operating system involved, and must be prepared for development sessions filled with obscure bugs which crash not just the protocol package but the entire operating system.

A final problem with processing at interrupt time is that the system scheduler has no control over the percentage of system time used by the protocol handler. If a large number of packets arrive, from a foreign host that is either malfunctioning or fast, all of the time may be spent in the interrupt handler, effectively killing the system.

There are other problems associated with putting protocols into an operating system kernel. The simplest problem often encountered is that the kernel address space is simply too small to hold the piece of code in question. This is a rather artificial sort of problem, but it is a severe problem none the less in many machines. It is an appallingly unpleasant experience to do an implementation with the knowledge that

for every byte of new feature put in one must find some other byte of old feature to throw out. It is hopeless to expect an effective and general implementation under this kind of constraint. Another problem is that the protocol package, once it is thoroughly entwined in the operating system, may need to be redone every time the operating system changes. If the protocol and the operating system are not maintained by the same group, this makes maintenance of the protocol package a perpetual headache.

The third option for protocol implementation is to take the protocol package and move it outside the machine entirely, on to a separate processor dedicated to this kind of task. Such a machine is often described as a communications processor or a front-end processor. There are several advantages to this approach. First, the operating system on the communications processor can be tailored for precisely this kind of task. This makes the job of implementation much easier. Second, one does not need to redo the task for every machine to which the protocol is to be added. It may be possible to reuse the same front-end machine on different host computers. Since the task need not be done as many times, one might hope that more attention could be paid to doing it right. Given a careful implementation in an environment which is optimized for this kind of task, the resulting package should turn out to be very efficient. Unfortunately, there are also problems with this approach. There is, of course, a financial problem associated with buying an additional computer. In many cases, this is not a problem at all since the cost is negligible compared to what the programmer would cost to do the job in the mainframe itself. More

fundamentally, the communications processor approach does not completely sidestep any of the problems raised above. The reason is that the communications processor, since it is a separate machine, must be attached to the mainframe by some mechanism. Whatever that mechanism, code is required in the mainframe to deal with it. It can be argued that the program to deal with the communications processor is simpler than the program to implement the entire protocol package. Even if that is so, the communications processor interface package is still a protocol in nature, with all of the same structural problems. Thus, all of the issues raised above must still be faced. In addition to those problems, there are some other, more subtle problems associated with an outboard implementation of a protocol. We will return to these problems later.

There is a way of attaching a communications processor to a mainframe host which sidesteps all of the mainframe implementation problems, which is to use some preexisting interface on the host machine as the port by which a communications processor is attached. This strategy is often used as a last stage of desperation when the software on the host computer is so intractable that it cannot be changed in any way. Unfortunately, it is almost inevitably the case that all of the available interfaces are totally unsuitable for this purpose, so the result is unsatisfactory at best. The most common way in which this form of attachment occurs is when a network connection is being used to mimic local teletypes. In this case, the front-end processor can be attached to the mainframe by simply providing a number of wires out of the front-end processor, each corresponding to a connection, which are

plugged into teletype ports on the mainframe computer. (Because of the appearance of the physical configuration which results from this arrangement, Michael Padlipsky has described this as the "milking machine" approach to computer networking.) This strategy solves the immediate problem of providing remote access to a host, but it is extremely inflexible. The channels being provided to the host are restricted by the host software to one purpose only, remote login. It is impossible to use them for any other purpose, such as file transfer or sending mail, so the host is integrated into the network environment in an extremely limited and inflexible manner. If this is the best that can be done, then it should be tolerated. Otherwise, implementors should be strongly encouraged to take a more flexible approach.

4. Protocol Layering

The previous discussion suggested that there was a decision to be made as to where a protocol ought to be implemented. In fact, the decision is much more complicated than that, for the goal is not to implement a single protocol, but to implement a whole family of protocol layers, starting with a device driver or local network driver at the bottom, then IP and TCP, and eventually reaching the application specific protocol, such as Telnet, FTP and SMTP on the top. Clearly, the bottommost of these layers is somewhere within the kernel, since the physical device driver for the net is almost inevitably located there. Equally clearly, the top layers of this package, which provide the user his ability to perform the remote login function or to send mail, are not entirely contained within the kernel. Thus, the question is not

whether the protocol family shall be inside or outside the kernel, but how it shall be sliced in two between that part inside and that part outside.

Since protocols come nicely layered, an obvious proposal is that one of the layer interfaces should be the point at which the inside and outside components are sliced apart. Most systems have been implemented in this way, and many have been made to work quite effectively. One obvious place to slice is at the upper interface of TCP. Since TCP provides a bidirectional byte stream, which is somewhat similar to the I/O facility provided by most operating systems, it is possible to make the interface to TCP almost mimic the interface to other existing devices. Except in the matter of opening a connection, and dealing with peculiar failures, the software using TCP need not know that it is a network connection, rather than a local I/O stream that is providing the communications function. This approach does put TCP inside the kernel, which raises all the problems addressed above. It also raises the problem that the interface to the IP layer can, if the programmer is not careful, become excessively buried inside the kernel. It must be remembered that things other than TCP are expected to run on top of IP. The IP interface must be made accessible, even if TCP sits on top of it inside the kernel.

Another obvious place to slice is above Telnet. The advantage of slicing above Telnet is that it solves the problem of having remote login channels emulate local teletype channels. The disadvantage of putting Telnet into the kernel is that the amount of code which has now

been included there is getting remarkably large. In some early implementations, the size of the network package, when one includes protocols at the level of Telnet, rivals the size of the rest of the supervisor. This leads to vague feelings that all is not right.

Any attempt to slice through a lower layer boundary, for example between internet and TCP, reveals one fundamental problem. The TCP layer, as well as the IP layer, performs a demultiplexing function on incoming datagrams. Until the TCP header has been examined, it is not possible to know for which user the packet is ultimately destined. Therefore, if TCP, as a whole, is moved outside the kernel, it is necessary to create one separate process called the TCP process, which performs the TCP multiplexing function, and probably all of the rest of TCP processing as well. This means that incoming data destined for a user process involves not just a scheduling of the user process, but scheduling the TCP process first.

This suggests an alternative structuring strategy which slices through the protocols, not along an established layer boundary, but along a functional boundary having to do with demultiplexing. In this approach, certain parts of IP and certain parts of TCP are placed in the kernel. The amount of code placed there is sufficient so that when an incoming datagram arrives, it is possible to know for which process that datagram is ultimately destined. The datagram is then routed directly to the final process, where additional IP and TCP processing is performed on it. This removes from the kernel any requirement for timer based actions, since they can be done by the process provided by the

user. This structure has the additional advantage of reducing the amount of code required in the kernel, so that it is suitable for systems where kernel space is at a premium. The RFC 814, titled "Names, Addresses, Ports, and Routes," discusses this rather orthogonal slicing strategy in more detail.

A related discussion of protocol layering and multiplexing can be found in Cohen and Postel [1].

5. Breaking Down the Barriers

In fact, the implementor should be sensitive to the possibility of even more peculiar slicing strategies in dividing up the various protocol layers between the kernel and the one or more user processes. The result of the strategy proposed above was that part of TCP should execute in the process of the user. In other words, instead of having one TCP process for the system, there is one TCP process per connection. Given this architecture, it is not longer necessary to imagine that all of the TCPs are identical. One TCP could be optimized for high throughput applications, such as file transfer. Another TCP could be optimized for small low delay applications such as Telnet. In fact, it would be possible to produce a TCP which was somewhat integrated with the Telnet or FTP on top of it. Such an integration is extremely important, for it can lead to a kind of efficiency which more traditional structures are incapable of producing. Earlier, this paper pointed out that one of the important rules to achieving efficiency was to send the minimum number of packets for a given amount of data. The idea of protocol layering interacts very strongly (and poorly) with this

goal, because independent layers have independent ideas about when packets should be sent, and unless these layers can somehow be brought into cooperation, additional packets will flow. The best example of this is the operation of server telnet in a character at a time remote echo mode on top of TCP. When a packet containing a character arrives at a server host, each layer has a different response to that packet. TCP has an obligation to acknowledge the packet. Either server telnet or the application layer above has an obligation to echo the character received in the packet. If the character is a Telnet control sequence, then Telnet has additional actions which it must perform in response to the packet. The result of this, in most implementations, is that several packets are sent back in response to the one arriving packet. Combining all of these return messages into one packet is important for several reasons. First, of course, it reduces the number of packets being sent over the net, which directly reduces the charges incurred for many common carrier tariff structures. Second, it reduces the number of scheduling actions which will occur inside both hosts, which, as was discussed above, is extremely important in improving throughput.

The way to achieve this goal of packet sharing is to break down the barrier between the layers of the protocols, in a very restrained and careful manner, so that a limited amount of information can leak across the barrier to enable one layer to optimize its behavior with respect to the desires of the layers above and below it. For example, it would represent an improvement if TCP, when it received a packet, could ask the layer above whether or not it would be worth pausing for a few milliseconds before sending an acknowledgement in order to see if the

upper layer would have any outgoing data to send. Dallying before sending the acknowledgement produces precisely the right sort of optimization if the client of TCP is server Telnet. However, dallying before sending an acknowledgement is absolutely unacceptable if TCP is being used for file transfer, for in file transfer there is almost never data flowing in the reverse direction, and the delay in sending the acknowledgement probably translates directly into a delay in obtaining the next packets. Thus, TCP must know a little about the layers above it to adjust its performance as needed.

It would be possible to imagine a general purpose TCP which was equipped with all sorts of special mechanisms by which it would query the layer above and modify its behavior accordingly. In the structures suggested above, in which there is not one but several TCPs, the TCP can simply be modified so that it produces the correct behavior as a matter of course. This structure has the disadvantage that there will be several implementations of TCP existing on a single machine, which can mean more maintenance headaches if a problem is found where TCP needs to be changed. However, it is probably the case that each of the TCPs will be substantially simpler than the general purpose TCP which would otherwise have been built. There are some experimental projects currently under way which suggest that this approach may make designing of a TCP, or almost any other layer, substantially easier, so that the total effort involved in bringing up a complete package is actually less if this approach is followed. This approach is by no means generally accepted, but deserves some consideration.

The general conclusion to be drawn from this sort of consideration is that a layer boundary has both a benefit and a penalty. A visible layer boundary, with a well specified interface, provides a form of isolation between two layers which allows one to be changed with the confidence that the other one will not stop working as a result. However, a firm layer boundary almost inevitably leads to inefficient operation. This can easily be seen by analogy with other aspects of operating systems. Consider, for example, file systems. A typical operating system provides a file system, which is a highly abstracted representation of a disk. The interface is highly formalized, and presumed to be highly stable. This makes it very easy for naive users to have access to disks without having to write a great deal of software. The existence of a file system is clearly beneficial. On the other hand, it is clear that the restricted interface to a file system almost inevitably leads to inefficiency. If the interface is organized as a sequential read and write of bytes, then there will be people who wish to do high throughput transfers who cannot achieve their goal. If the interface is a virtual memory interface, then other users will regret the necessity of building a byte stream interface on top of the memory mapped file. The most objectionable inefficiency results when a highly sophisticated package, such as a data base management package, must be built on top of an existing operating system. Almost inevitably, the implementors of the database system attempt to reject the file system and obtain direct access to the disks. They have sacrificed modularity for efficiency.

The same conflict appears in networking, in a rather extreme form.

The concept of a protocol is still unknown and frightening to most naive programmers. The idea that they might have to implement a protocol, or even part of a protocol, as part of some application package, is a dreadful thought. And thus there is great pressure to hide the function of the net behind a very hard barrier. On the other hand, the kind of inefficiency which results from this is a particularly undesirable sort of inefficiency, for it shows up, among other things, in increasing the cost of the communications resource used up to achieve the application goal. In cases where one must pay for one's communications costs, they usually turn out to be the dominant cost within the system. Thus, doing an excessively good job of packaging up the protocols in an inflexible manner has a direct impact on increasing the cost of the critical resource within the system. This is a dilemma which will probably only be solved when programmers become somewhat less alarmed about protocols, so that they are willing to weave a certain amount of protocol structure into their application program, much as application programs today weave parts of database management systems into the structure of their application program.

An extreme example of putting the protocol package behind a firm layer boundary occurs when the protocol package is relegated to a front-end processor. In this case the interface to the protocol is some other protocol. It is difficult to imagine how to build close cooperation between layers when they are that far separated. Realistically, one of the prices which must be associated with an implementation so physically modularized is that the performance will suffer as a result. Of course, a separate processor for protocols could be very closely integrated into

the mainframe architecture, with interprocessor co-ordination signals, shared memory, and similar features. Such a physical modularity might work very well, but there is little documented experience with this closely coupled architecture for protocol support.

6. Efficiency of Protocol Processing

To this point, this document has considered how a protocol package should be broken into modules, and how those modules should be distributed between free standing machines, the operating system kernel, and one or more user processes. It is now time to consider the other half of the efficiency question, which is what can be done to speed the execution of those programs that actually implement the protocols. We will make some specific observations about TCP and IP, and then conclude with a few generalities.

IP is a simple protocol, especially with respect to the processing of normal packets, so it should be easy to get it to perform efficiently. The only area of any complexity related to actual packet processing has to do with fragmentation and reassembly. The reader is referred to RFC 815, titled "IP Datagram Reassembly Algorithms", for specific consideration of this point.

Most costs in the IP layer come from table look up functions, as opposed to packet processing functions. An outgoing packet requires two translation functions to be performed. The internet address must be translated to a target gateway, and a gateway address must be translated to a local network number (if the host is attached to more than one

network). It is easy to build a simple implementation of these table look up functions that in fact performs very poorly. The programmer should keep in mind that there may be as many as a thousand network numbers in a typical configuration. Linear searching of a thousand entry table on every packet is extremely unsuitable. In fact, it may be worth asking TCP to cache a hint for each connection, which can be handed down to IP each time a packet is sent, to try to avoid the overhead of a table look up.

TCP is a more complex protocol, and presents many more opportunities for getting things wrong. There is one area which is generally accepted as causing noticeable and substantial overhead as part of TCP processing. This is computation of the checksum. It would be nice if this cost could be avoided somehow, but the idea of an end-to-end checksum is absolutely central to the functioning of TCP. No host implementor should think of omitting the validation of a checksum on incoming data.

Various clever tricks have been used to try to minimize the cost of computing the checksum. If it is possible to add additional microcoded instructions to the machine, a checksum instruction is the most obvious candidate. Since computing the checksum involves picking up every byte of the segment and examining it, it is possible to combine the operation of computing the checksum with the operation of copying the segment from one location to another. Since a number of data copies are probably already required as part of the processing structure, this kind of sharing might conceivably pay off if it didn’t cause too much trouble to

the modularity of the program. Finally, computation of the checksum seems to be one place where careful attention to the details of the algorithm used can make a drastic difference in the throughput of the program. The Multics system provides one of the best case studies of this, since Multics is about as poorly organized to perform this function as any machine implementing TCP. Multics is a 36-bit word machine, with four 9-bit bytes per word. The eight-bit bytes of a TCP segment are laid down packed in memory, ignoring word boundaries. This means that when it is necessary to pick up the data as a set of 16-bit units for the purpose of adding them to compute checksums, horrible masking and shifting is required for each 16-bit value. An early version of a program using this strategy required 6 milliseconds to checksum a 576-byte segment. Obviously, at this point, checksum computation was becoming the central bottleneck to throughput. A more careful recoding of this algorithm reduced the checksum processing time to less than one millisecond. The strategy used was extremely dirty. It involved adding up carefully selected words of the area in which the data lay, knowing that for those particular words, the 16-bit values were properly aligned inside the words. Only after the addition had been done were the various sums shifted, and finally added to produce the eventual checksum. This kind of highly specialized programming is probably not acceptable if used everywhere within an operating system. It is clearly appropriate for one highly localized function which can be clearly identified as an extreme performance bottleneck.

Another area of TCP processing which may cause performance problems is the overhead of examining all of the possible flags and options which

occur in each incoming packet. One paper, by Bunch and Day [2], asserts that the overhead of packet header processing is actually an important limiting factor in throughput computation. Not all measurement experiments have tended to support this result. To whatever extent it is true, however, there is an obvious strategy which the implementor ought to use in designing his program. He should build his program to optimize the expected case. It is easy, especially when first designing a program, to pay equal attention to all of the possible outcomes of every test. In practice, however, few of these will ever happen. A TCP should be built on the assumption that the next packet to arrive will have absolutely nothing special about it, and will be the next one expected in the sequence space. One or two tests are sufficient to determine that the expected set of control flags are on. (The ACK flag should be on; the Push flag may or may not be on. No other flags should be on.) One test is sufficient to determine that the sequence number of the incoming packet is one greater than the last sequence number received. In almost every case, that will be the actual result. Again, using the Multics system as an example, failure to optimize the case of receiving the expected sequence number had a detectable effect on the performance of the system. The particular problem arose when a number of packets arrived at once. TCP attempted to process all of these packets before awaking the user. As a result, by the time the last packet arrived, there was a threaded list of packets which had several items on it. When a new packet arrived, the list was searched to find the location into which the packet should be inserted. Obviously, the list should be searched from highest sequence number to lowest sequence

number, because one is expecting to receive a packet which comes after those already received. By mistake, the list was searched from front to back, starting with the packets with the lowest sequence number. The amount of time spent searching this list backwards was easily detectable in the metering measurements.

Other data structures can be organized to optimize the action which is normally taken on them. For example, the retransmission queue is very seldom actually used for retransmission, so it should not be organized to optimize that action. In fact, it should be organized to optimized the discarding of things from it when the acknowledgement arrives. In many cases, the easiest way to do this is not to save the packet at all, but to reconstruct it only if it needs to be retransmitted, starting from the data as it was originally buffered by the user.

There is another generality, at least as important as optimizing the common case, which is to avoid copying data any more times than necessary. One more result from the Multics TCP may prove enlightening here. Multics takes between two and three milliseconds within the TCP layer to process an incoming packet, depending on its size. For a 576-byte packet, the three milliseconds is used up approximately as follows. One millisecond is used computing the checksum. Six hundred microseconds is spent copying the data. (The data is copied twice, at .3 milliseconds a copy.) One of those copy operations could correctly be included as part of the checksum cost, since it is done to get the data on a known word boundary to optimize the checksum algorithm.

However, the copy also performs another necessary transfer at the same time. Header processing and packet resequencing takes .7 milliseconds. The rest of the time is used in miscellaneous processing, such as removing packets from the retransmission queue which are acknowledged by this packet. Data copying is the second most expensive single operation after data checksuming. Some implementations, often because of an excessively layered modularity, end up copying the data around a great deal. Other implementations end up copying the data because there is no shared memory between processes, and the data must be moved from process to process via a kernel operation. Unless the amount of this activity is kept strictly under control, it will quickly become the major performance bottleneck.

7. Conclusions

This document has addressed two aspects of obtaining performance from a protocol implementation, the way in which the protocol is layered and integrated into the operating system, and the way in which the detailed handling of the packet is optimized. It would be nice if one or the other of these costs would completely dominate, so that all of one’s attention could be concentrated there. Regrettably, this is not so. Depending on the particular sort of traffic one is getting, for example, whether Telnet one-byte packets or file transfer maximum size packets at maximum speed, one can expect to see one or the other cost being the major bottleneck to throughput. Most implementors who have studied their programs in an attempt to find out where the time was going have reached the unsatisfactory conclusion that it is going

equally to all parts of their program. With the possible exception of checksum processing, very few people have ever found that their performance problems were due to a single, horrible bottleneck which they could fix by a single stroke of inventive programming. Rather, the performance was something which was improved by painstaking tuning of the entire program.

Most discussions of protocols begin by introducing the concept of layering, which tends to suggest that layering is a fundamentally wonderful idea which should be a part of every consideration of protocols. In fact, layering is a mixed blessing. Clearly, a layer interface is necessary whenever more than one client of a particular layer is to be allowed to use that same layer. But an interface, precisely because it is fixed, inevitably leads to a lack of complete understanding as to what one layer wishes to obtain from another. This has to lead to inefficiency. Furthermore, layering is a potential snare in that one is tempted to think that a layer boundary, which was an artifact of the specification procedure, is in fact the proper boundary to use in modularizing the implementation. Again, in certain cases, an architected layer must correspond to an implemented layer, precisely so that several clients can have access to that layer in a reasonably straightforward manner. In other cases, cunning rearrangement of the implemented module boundaries to match with various functions, such as the demultiplexing of incoming packets, or the sending of asynchronous outgoing packets, can lead to unexpected performance improvements compared to more traditional implementation strategies. Finally, good performance is something which is difficult to retrofit onto an existing

program. Since performance is influenced, not just by the fine detail, but by the gross structure, it is sometimes the case that in order to obtain a substantial performance improvement, it is necessary to completely redo the program from the bottom up. This is a great disappointment to programmers, especially those doing a protocol implementation for the first time. Programmers who are somewhat inexperienced and unfamiliar with protocols are sufficiently concerned with getting their program logically correct that they do not have the capacity to think at the same time about the performance of the structure they are building. Only after they have achieved a logically correct program do they discover that they have done so in a way which has precluded real performance. Clearly, it is more difficult to design a program thinking from the start about both logical correctness and performance. With time, as implementors as a group learn more about the appropriate structures to use for building protocols, it will be possible to proceed with an implementation project having more confidence that the structure is rational, that the program will work, and that the program will work well. Those of us now implementing protocols have the privilege of being on the forefront of this learning process. It should be no surprise that our programs sometimes suffer from the uncertainty we bring to bear on them.

Citations

[1] Cohen and Postel, "On Protocol Multiplexing", Sixth Data Communications Symposium, ACM/IEEE, November 1979.

[2] Bunch and Day, "Control Structure Overhead in TCP", Trends and Applications: Computer Networking, NBS Symposium, May 1980.