AN ANALYSIS OF TCP PROCESSING OVERHEAD

David D. Clark
Massachusetts Institute of Technology
Cambridge, MA

John Romkey
Epilogue, Inc.
Belmont, CA

Howard Salwen
Proteon, Inc.
Westboro, MA

Abstract

This paper reports a preliminary analysis of the processing overhead of the transport protocol TCP, done to estimate the possible performance range of the protocol. The analysis was performed by compiling a version of TCP and counting the number of the instructions in the common path. The analysis suggests that fewer than 200 instructions are required to process a TCP packet in the normal case. This number is small enough to support very high-speed transmission if it were the major overhead. The paper offers some speculation about the actual source of processing overhead in network protocols.

1. Introduction

While networks, especially local area networks, have been getting faster, perceived throughput at the application has not always increased accordingly. Various performance bottlenecks have have been encountered, each of which has to be analyzed and corrected.

One aspect of networking often suspected of contributing to low throughput is the transport layer of the protocol suite. This layer, especially in connectionless protocols, has considerable functionality, and is typically executed in software by the host processor at the end points of the network. It is thus a likely source of processing overhead.

While this theory is appealing, a preliminary examination suggested to us that other aspects of networking may be a more serious source of overhead. To test this proposition, a detailed study was made of a popular transport protocol, TCP[1]. This paper provides preliminary results of that study. Our tentative conclusion is that TCP is in fact not the source of the overhead often observed in packet processing, and that if properly implemented, could support very high speeds.

2. TCP

TCP, or Transmission Control Protocol, is the transport protocol from the Internet protocol suite. The Internet protocols are *connectionless*, so the network layer has minimal function. The functions of detecting and recovering lost or corrupted packets, flow control, and multiplexing are performed at the transport level. TCP uses sequence numbers, cumulative acknowledgment, windows and software checksums to implement these functions.

TCP is used on top of a network level protocol called Internet, or IP[2]. This protocol, which is a datagram packet delivery protocol, deals with host addressing and routing, but that latter function is almost totally the task of the Internet level packet switch, or *gateway*. IP also provides the ability for packets to be broken into smaller units, or *fragmented*, on passing into a network with a smaller maximum packet size. The IP layer at the receiving end is responsible for reassembling these fragments. For a general review of TCP and IP, see[3] or[4].

Under IP is the layer dealing with the specific network technology being used. This may be a very simple layer in the case of a local area network, or a rather complex layer for a network such as X.25. On top of TCP sits one of a number of application protocols, most commonly for remote login, file transfer or mail.

3. The Analysis

This study addressed the overhead of running TCP and IP (since TCP is never run without IP), and the overhead of the operating system support needed by them. It did not consider the cost of the driver for some specific network, nor did it consider the cost of running an application.

The study technique is very simple: we compiled a TCP, identified the normal path through the code, and counted the instructions. However, more detail is required to put our work in context.

The TCP we used is the currently distributed version of TCP for Unix from Berkeley[5]. By using a production quality TCP, we believe that we can avoid the charge that our TCP is not fully functional.

While we used a production TCP as a starting point for our analysis, we made significant changes to the code. To give TCP itself a fair hearing, we felt it was necessary to pry it apart from some protocol-independent overheads with which TCP is closely associated, especially in this implementation.

Chief among these is the data buffering technique for the Berkeley TCP. In this implementation, data is stored in a series of chained buffers, called *m-bufs*. This technique of chaining small buffers together was used to avoid the excessive waste that can occur with a simpler scheme which allocates a maximum size buffer for each packet. While m-bufs certainly avoid the problem of internal buffer fragmentation, there is a considerable overhead possibly associated with managing these buffers.

We felt that this buffering scheme, as well as the scheme for managing timers and other system features, was a characteristic of Unix rather then the TCP, and it was reasonable to separate the cost of TCP from the cost of these support functions. At the same time, we wanted our evaluation to be realistic. So it was not fair to ignore altogether the cost of these functions.

Our approach was to take the Berkeley TCP as a starting point, and modify it to better give a measure of intrinsic costs. One of us (Romkey) removed from the TCP code all references to Unix-specific functions such as m-bufs, and replaced them with working but specialized versions of the same functions. To insure that the resulting code was still operational, it was compiled and executed. Running the TCP in two Unix address spaces, and passing packets by an interprocess communication path, the TCP was made to open and close connections, and to pass data. While we did not test the TCP against other implementations, we can be reasonably certain that the TCP that resulted from our test was essentially a correctly implemented TCP.

The compiler used for this experiment generated reasonably efficient code for the Intel 80386. Other experiments we have performed tend to suggest that for this sort of application, the number of instructions is very similar for a 80386, a Motorola 68020, or even a RISC chip such as the SPARC.

4. Finding the Common Path

One observation central to the efficient implementation of TCP is to observe that while there are many paths through the code, there is only one common one. While opening or closing the connection, or after errors, special code will be executed. But none of this code is required for the normal case. The normal case is data transfer, while the TCP connection is established. In this state, data flows in one direction, and acknowledgment and window information flows in the other.

In writing a TCP, it is important to optimize this path. In studying the TCP, it was necessary to find and follow it in the code. Since the Berkeley TCP did not separate this path from all the other cases, we were not sure if it was being executed as efficiently as possible. For this reason, and to permit a more direct analysis, we implemented a special "fast path" TCP. When a packet was received, some simple tests were performed, to see if the connection was in established state, if the packet had no special control flags on, and if the sequence number was expected. If so, control was transferred to the fast path. The version of the TCP which we compiled and tested had this fast path, and it was this fast path we audited.

There are actually two common paths through the TCP, the sending end and the receiving end. In general, TCP permits both ends to do both at once, although in the real world it happens only in some limited cases. But in any bulk data example, where throughput is an issue, data almost always flows in only one direction. One end, the sending end, puts data in its outgoing packets. When it receives a packet, it finds only control information: acknowledgments and windows.

The other, receiving, end, finds data in its incoming packets and sends back control information. In this paper we will use these two terms, sender and receiver, to describe the direction of data flow. Both ends really do receive packets, but only one end tends to receive data.

5. A First Case Study -- Input Processing

A common belief about TCP is that the most complex, and thus most costly part, is the packet receiving operation. When receiving a packet, the program must proceed through the packet testing each field for errors and determining the proper action to take. In contrast, when sending a packet, the program knows exactly what actions are intended and has essentially to format the packet and start the transmission.

A preliminary investigation tended to support this model, and so for our first detailed analysis, we studied the program which receives and processes a packet.

There are three general stages to the TCP processing. In the first, the TCP checksum is verified. This requires computing a simple function of all the bytes in the packet. In the second, a search is made to find the local state information (called the transmission control block, or TCB) for this TCP connection. In the third stage, the packet header is processed.

We chose not to study these first two stages. The checksum cost depends strongly on the raw speed of the environment and the detailed coding of the computation. The lookup function similarly depends on the details of the data structure, the assumed number of connections, and the potential for special hardware. In a later section, we will return to these two operations, but in the detailed analysis of this section, they are omitted.

The following analysis thus covers the TCP processing from the point where the packet has been checksummed, and the TCB has been found. It covers the processing of all the header data, and the resulting actions.

The packet input processing code has a rather different path for the sender and for the receiver of data.

The overall numbers are the following:

- Sender of data-- 191 to 235 instructions.
- Receiver of data -- 186 instructions.

A more detailed breakdown provides further insight.

Both sides contain a common path of 154 instructions. Of these, 15 are procedure entry and exit or initialization. For the receiver of data, an additional 15 instructions are spent sequencing the data and calling the buffer manager, and another 17 are spent processing the window field in the packet.

The sender of data, which is receiving control information, has more steps to perform. In addition to the 154 common instructions, it takes 9 to process the acknowledgment, 20 to process the window, 17 to compute the outgoing congestion window (so-called "slow-start" control), and 44 instructions (but not for each packet) to estimate the round-trip time. The round-trip delay is measured not for every packet, but only once per round trip. For short delay paths, where one packet can be sent in one round-trip, this cost could occur for every packet. For longer paths, the cost will be spread over several packets.

From this level of analysis of one part of the code, it is possible to draw a number of conclusions. First, there are actually very few instructions required. In the process of making this study, we found several opportunities for shortening the path length. None of those are in this version.

Second, a significant amount of code is involved in control of the protocol dynamics; computing the congestion window and the round trip delay. These activities have nothing to do with the actual data flow. The actual management of the sequence numbers is very quick. But between 17 and 61 instructions are spent on computation of dynamic control parameters.

The analysis made clear that some changes to the protocol would provide a slight speedup. But these changes are not what is often proposed.

One change concerns sequence numbers. TCP provides a sequence number for data, but not for individual packets. Control information is thus sequenced only indirectly, an approach which is functionally correct but which generates overhead. If every packet had a sequence number, 15 instructions could be removed from the processing of the window information. A reduction would also probably occur in the estimation of the round-trip delay.

In a later section, we will return to a more global speculation on what these numbers mean.

6. TCP Output Processing

We subjected the output side of TCP to an analysis that was somewhat less detailed than the input side. We did not program a fast path, and we did not attempt to separate the paths for data sending and receiving. We thus have a single number that is a combination of the two paths, and which (by inspection) could be significantly improved by an optimization of the common path.

We found 235 instructions to send a packet in TCP. While this number is actually greater than the cost of receiving a packet, we believe that the lack of optimization in this code precludes direct comparison with input processing. With equivalent attention to the code, the output side should be less costly than the input side.

7. The Cost of IP

In the normal case, IP performs very few functions. On input of a packet, it checks the header for correct form, extracts the protocol number, and call the TCP processing function. The executed path is almost always the same. On output, the operation is even more simple.

The instruction counts for IP were as follows:

- Packet receipt -- 57
- Packet sending -- 61

8. Output Processing -- An Implementation Trick

Output processing seems much less complex than input. There is no question of testing for mal-formed packets, or looking up a TCB. The TCB is known, as is the desired action.

To take advantage of this constrained environment, we programmed an optimization as part of the IP output routine.

IP places a fixed size 20 byte header on the front of every IP packet, plus a variable amount of options. Most IP packets carry no options. Of the 20 byte header, 14 of the bytes will be the same for all IP packets sent by a particular TCP connection. The IP length, ID and checksum fields (6 bytes total) will probably be different for each packet. Also, if a packet carries any options, all packets for that TCP connection will likely carry the same options.

Based on this observation, we designed an IP layer which created a template IP header with the constant fields filled in, and associated this template with a TCP connection. When TCP wished to send a packet on that connection, it would call IP, passing it the template and the length of the packet, and IP would block copy the template into the space for the IP header, fill in the length field, fill in the unique ID field and calculate the IP header checksum.

The IP layer is also responsible for routing packets. Discovering a route to a host is an expensive operation which we do not want to perform on every packet, so we cache the route to a particular host. The route may change, however, and invalidating the cache can also be a tricky problem. We want to make sure that routing cache lookups are as fast as possible, and while invalidating the cache may be much slower without impacting overall TCP throughput, they should still be relatively efficient.

The solution is to have each TCP connection have a timestamped cached routing entry. Routing entries are invalidated when an IP router sends an ICMP redirect (redirecting the IP to use a different router) to this host. The IP layer should then store the host/router pair contained in the redirect in a routing table and set an internal variable to the time that the redirect was received. Then when TCP asks the IP layer to transmit a packet, IP checks if the time the route was cached before the last ICMP redirect was received (which might have changed this route). If it was, the route is recomputed (with information from the stored routing table) and recached; otherwise the cached route is used.

There may also be an entry point into the IP layer which TCP may use to request that a connection be rerouted (in case of excessive retransmissions, for instance). This function fits in easily with the caching model, but our TCP/IP implementation did not use or provide such a function.

9. Support Functions

9.1. The Buffer Layer

The most complex of the support functions is the layer that manages the buffers which hold the data at the interface to the layer above. Our buffer layer was designed to match high-throughput bulk data transfer. It supports a allocate and free function, and a simple get and put interface with one additional feature to support data sent but not yet acknowledged. All the bookkeeping about out of order packets was performed by TCP itself.

The buffer layer added the following costs to the processing of a packet.

- Sending a data packet -- 40 instructions.
- Receiving a data packet -- 35 instructions .
- Receiving an acknowledgment (may free a buffer) -- 30 instructions.

It might be argued that our buffer layer is too simple. We would accept that argument, but are not too concerned by it. All transport protocols must have a buffer layer. In comparing two transport protocols, it is reasonable to assume (to first order) that if they have equivalent service goals they will have equivalent buffer layers.

A buffer layer can easily grow in complexity to swamp the protocol itself. The reason for this is that the buffer layer is that part of the code in which the demand for varieties of service has a strong effect. For example, some implementations of TCP attempt to provide good service to application clients that want to deal with data one byte at a time, as well as others that want to deal in large blocks. To serve both sorts of clients requires a buffer layer complex enough to fold both of these models together. In an informal study done by one of us (Clark) of another transport protocol, an extreme version of this problem was uncovered: of 68 pages of code written in C, which seemed to be the transport protocol, over 60 were found to be the buffer layer and interfaces to other protocol layers, and only about 6 were the protocol.

The problem of the buffer layer is made worse by the fact that the protocol specifiers do not admit that such a layer exists. It is not a part of the ISO reference model, but is left as an exercise for the implementor. This is reasonable, within limits, since the design of the buffer layer has much to do with the particular operating system. (This, in fact, contributed to the great simplicity of our buffer layer; since there was no operating system to speak of, we were free to structure things as needed with the right degree of generality and functionality.)

However, some degree of guidance to the implementor is necessary, and the specifiers of a protocol suite would be well served to give some thought to the role of buffering in their architecture.

9.2. Timers and Schedulers

In TCP, almost every packet is coupled to a timer. On sending data, a retransmit timer is set. On receipt of an acknowledgment, this timer is cleared. On receiving data, a timer may be set to permit dallying before sending the acknowledgment. On sending the acknowledgment, if that timer has not expired it must be cleared.

The overhead of managing these timers can sometimes be a great burden. Some operating systems' designers did not think that timers would be used in this demanding a context, and made no effort to control their costs.

In this implementation, we used a specialized timer package similar to the one described by Varghese[6]. It provides a very low cost for timer operations. In our version the costs were:

- Set a timer -- 35
- Clear a timer -- 17

• Reset a timer (clear and set together) -- 41

10. Checksums and TCBs -- The Missing Steps

In the discussion of TCP input processing above, we intentionally omitted a cost for computing the TCP checksum, and for looking up the TCB. We now consider each of these costs.

The TCP checksum is a point of long-standing contention among protocol designers. Having an end-to-end checksum which is computed after the packet is actually in main memory provides a level of protection that is very valuable[7]. However, computing this checksum using the central processor rather than some outboard chip is a considerable burden on the protocol. In this paper we do not want to take sides on this matter. We only observe that "you get what you pay for." A protocol designer might try to make the cost optional, and should certainly design the checksum to be as efficient as possible. TCP does not do this.

There are a number of processing overheads associated with processing the bytes of the packet, rather then the header fields. The checksum computation is one of these, but there are others. In a later section we consider all the costs of processing the bytes.

Looking up the TCB is also a cost somewhat unrelated to the details of TCP. That is, any transport protocol must keep state information for each connection, and must use a search function to find this for an incoming packet. The only variation is the number of bits that must be matched to find the state (TCP uses 64, which may not be minimal), and the number of connections that are assumed to be open.

Using the principle of the common path, however, one can provide algorithms that are very cheap. While we have not yet coded them, they are plausible and easy to estimate.

The most simple algorithm is to assume that the next packet is from the same connection as the last packet. To check this, one need only pick up a pointer to the TCB saved from last time, extract from the packet and compare the correct 64 bits, and return the pointer. This takes less than 10 instructions. If this optimization fails too often to be useful, the next step is to hash the 64 bits into a smaller value, perhaps a 8 bit field, and use this to index into an array of linked lists of TCBs, with the most recently used TCB sorted first. If the needed TCB is indeed first on the list selected by the hash function, the cost is again very low. A reasonable estimate is 25 instructions. We will use this higher estimate in the analysis to follow.

11. Some Speed Predictions

Adding all these costs together, we see that the overhead of receiving a packet with control information in it (which is the most costly version of the processing path) is about 357 instructions. This includes the TCP and IP level processing, our crude estimate of the cost of finding the TCB, the buffer layer, and resetting a timer. Adding up the other versions of the sending and receiving paths yields instruction counts of the same magnitude.

With only minor optimization, an estimate of 300 instructions could be justified as a round number to use as a basis for some further analysis. If the processing overhead were the only bottleneck, how fast could a stream of TCP packets forward data?

Obviously, we must assume some target processor to estimate processing time. While these estimates were made for an Intel 80386, we believe the obvious processor is a 32 bit RISC chip, such as a SPARC chip or a Motorola 88000. A conservative execution rate for such a machine might be 10 mips, since chips of this sort can be expected to have a clock rate of twice that or more, and execute most instructions in one clock cycle. (The actual rate clearly requires a more detailed analysis: it depends on the number of data references, the data fetch architecture of the chip, the supporting memory architecture, and so on. For this paper, which is only making a very rough estimate, we believe that a working number of 10 mips is reasonable.)

In fairness, the estimate of 300 instructions should be adjusted for the change from the 80386 to a RISC instruction set. However, based on another study performed of packet processing code, we found little expansion of the code when converting to a RISC chip. The operations required for packet processing are so simple that no matter what processor is being used, the instruction set actually utilized is a RISC set.

A conservative adjustment would be to assume that 300 instructions for a 80386 would be 400 instructions for a RISC processor.

At 10 mips, a processor can execute 400 instruction in 40 us., or 25,000 packets per second. These processing costs permit rather high data rates.

If we assume a packet size of 4000 bytes, which would fit in an FDDI frame, for example, then 25,000 packets per second provides 800 megabits per second.

Figuring another way, if we assume an FDDI network with 100 megabits per second bandwidth, how small can the packets get before the processing per packet limits the throughput? The answer is 500 bytes.

These numbers are very encouraging. They suggest that it is not necessary to revise the protocols to utilize a network such as FDDI. It is only necessary to implement them properly.

12. Why Are Protocols Slow?

The numbers computed above may seem hard to believe. While the individual instruction counts may seem reasonable, the overall conclusion is not consistent with observed performance today.

We believe that the proper conclusion is that protocol processing is not the real source of the processing overhead. There are several others that are more important. They are just harder to find, and the TCP is easier to blame.

The first overhead is the operating system. As we discussed above, packet processing requires considerable support from the system. It is necessary to take an interrupt, allocate a packet buffer, free a packet buffer, restart the I/O device, wake up a process (or two or three), and reset a timer. In a particular implementation, there may be other costs that we did not identify in this study.

In a typical operating system, these functions may turn out to be very expensive. Unless they were designed for exactly this function, they may not match the performance requirements at all.

A common example is the timer package. Some timer packages are designed under the assumption that the common operations are setting a timer, and having a timer expire. These operations are made less costly, at the expense of the operation of unsetting, or clearing the timer. But that is what happens on every packet.

It may seem as if these functions, even if not optimized, are small compared to TCP. This is true only if TCP is big. But, as we discovered above, TCP is small. If a typical path through TCP is 200 instructions, a timer package could cost that much if not carefully designed.

The other major overhead in packet processing is performing operations that touch the bytes. The example associated with the transport protocol is computing the checksum. The more important one is moving the data in memory.

Data is moved in memory for two reasons. First, it is moved to separate the data from the header, and get the data into the alignment needed by the application. Second, it is copied to get it from I/O device to system address space to user address space.

In a good implementation, these operations will be combined to require a minimal number of copies. In the Berkeley Unix, for example, when receiving a packet, the data is moved from the I/O device into the chained m-buf structure, and then is moved into the user address space in a location that is aligned as the user needs it. The first copy may be done by DMA controller or by the processor, the second is always done by the processor.

It is harder than one would wish to get these copy operations to run fast. Typically, one must use two instructions to move 4 bytes. A 4000 byte packet thus requires 2000 instructions, almost 20 times as many as needed to process the header. Even if one instruction is needed, it must be complex, with two index increments, (not an option with a RISC processor) and there must be 2000 data references.

The checksum has a similar cost. While there are only half the data operations (the data is read but not written), there are probably two instructions per 4 bytes.

To avoid these costs, one might postulate a special controller, a relative of a DMA controller, that would perform memory-to-memory transfers without loading the processor, or perform checksums. This is a nice idea, but requires that the memory and bus of the computer have enough bandwidth to allow both the copy controller and the processor to run concurrently. This adds to the cost of the bus and the memory, which may not be justified unless the computer is to be optimized for network applications.

To copy data, one must use two memory cycles, a read and a write.. In other words, the bandwidth of the memory must be twice the achieved rate of the copy. Receiving a packet thus requires four memory cycles per word, one for the input DMA, one for the checksum and two for the copy.

A 32 bit memory with a cycle time of 250 ns., typical for dynamic RAMs today, would thus imply a memory limit of 32 megabits per second. This is a far more important limit than the TCP processing limits computed above. Our estimates of TCP overhead could be off by several factors of two before the overhead of TCP would intrude into the limitations of the memory.

13. Some Dangerous Speculations

If the operating system and the memory overhead are the real limits, it is tempting to avoid these by moving the processing outboard from the processor onto a special controller. This controller could run a specialized version of an operating system (similar to the one we postulated for this analysis) and could have a special high-performance memory architecture. By matching the memory to the special needs of packet processing, one could achieve high performance at acceptable cost.

For example, since almost all the memory cycles are sequential copy or checksum operations, one could perform these in a way that takes advantage of the high-speed sequential access methods now available in dynamic memory chips. These methods, variously called "page mode", or "static column mode", or "nibble mode", permit cycles of 40 or 50 ns. or faster, a great speedup over the normal cycle. Alternatively, one could use static memory, video RAM, or a more pipelined approach to achieve performance.

To our knowledge, this has not be attempted. There are products today that provide a outboard implementation of TCP. But they seem more intended to provide ease of use and portability rather than high performance.

One might try for both ease of use and performance. The problem with this is the buffer layer, that un-architected layer above TCP that passes the data to the application. If TCP runs in an outboard processor, then the interface to the host will be in terms of the buffer. If this is high performance, it will probably be complex. That is the reality of buffer layers in host-resident implementations; splitting it between host and outboard processor will almost certainly make it worse.

14. Observations

Since this paper is about TCP, it is appropriate to note aspects of the protocol that add to the processing burden. The TCP checksum, as we hinted above, is one such case. In TCP, a single checksum field is used to protect both the TCP header and the date itself. This means that the checksum of the data must be computed first, before doing any header processing.

In NETBLT[8], a protocol designed to optimize throughput, there are two checksum fields. One protects the header, the other the data. With this change, the header checksum is tested first, but the data checksum can be put off. It is put off, in fact, until the data is read from memory as part of the copy to remove the header. By combining the copy and checksum operation, one of the four memory operations is eliminated. A version of NETBLT (for the IBM AT) was actually coded this way, and the resulting version ran considerably faster that the one with a separate copy and checksum loop.

14.1. Protocols In Silicon

It has been proposed, for example by the designers of XTP[9], that to achieve reasonable throughput, it will be necessary to abandon protocols such as TCP and move to more efficient protocols that can be computed by hardware in special chips.

The designers of XTP must confront the problems discussed in this paper if they are to be successful in the quest for a high speed protocol processor. It is not enough to be better than TCP and to be compatible with the form and function of silicon technology. The protocol itself is a small fraction of the problem. The XTP protocol must still be interfaced with the host operating system and the rest of the environment.

Our analysis suggests that TCP can do a creditable job given the right environment. What is needed is to move to an efficient processing environment, such as a high-performance outboard processor card with special memory and controllers for byte operations such as copy and checksum. In this context, a fast general purpose processor can still be used to perform the protocol processing.

If high performance is possible with a programmable element using general protocols, it is highly desirable. The experience of the network community with TCP shows why. TCP is 15 years old this year. Yet we are still tinkering with it trying to get it right. The problem is not the data processing, but the algorithms that deal with the network dynamics. In our analysis, we found a significant part of the overhead was computing control parameters. The particular algorithm in our code was just developed in the last year[5], 15 years after the first TCP proposal, and we can expect further changes with further experience.

As we move to higher rates, we can expect similar experiences. These aspects of the protocol must not be cast in silicon, or we risk having something that cannot be made to work well, which is the primary goal that drives us to silicon.

Our analysis suggests that putting protocols in hardware is not required. While a special processing environment will be needed, we can still use standard protocols and programmable controllers.

References

1. Information Sciences Institute, "Transmission Control Protocol NIC-RFC 793", *DDN Protocol Handbook*,Vol. 2September 1981, pp. 2.179-2.198.

2. Information Sciences Institute, "DARPA Internet Program Protocol Specification NIC-RFC 791", *DDN Protocol Handbook*,Vol. 2September 1981, pp. 2.99-2.149.

3. Postel, J.B., Sunshine, C.A., Cohen, D., "The ARPA Internet Protocol", *Computer Networks 5*,Vol. 5No. 4July 1981, pp. 261-271.

4. Postel, Jonathan. B., "Internetwork Protocol Approaches", *IEEE Transactions on Communications*,Vol. Com-28No. 4April 1980, pp. 605-611.

5. Jacobson, V., "Congestion Avoidance and Control", Tech. report, Lawrence Berkeley Laboratory, 1988.

6. Varghese, G. and Lauck, T., "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility", *Proceedings of the Eleventh ACM Symposium on Operating Systems Principles*, ACM Operating Systems Review, Austin, TX, November 1987.

7. Saltzer, J.H., Reed, D.P., Clark, D.D., "End-to-End Arguments in System Design", *ACM Transactions on Computer Systems*,Vol. 2No. 4November 1984, pp. 277-288.

8. Clark, D., Lambert, M., Zhang, L., "NETBLT: A High Throughput Transport Protocol", *Frontiers in Computer Communications Technology: Proceedings of the ACM-SIGCOMM '87*, Association for Computing Machinery, Stowe, VT, August 1987, pp. 353-359.

9. Chesson, G., Eich, B., Schryver, V., Cherenson, A., and Whaley, A., "XTP Protocol Definition", Tech. report Revision 3.0, Silicon Graphics, Inc., January 1988.