An Analysis of TCP Processing Overhead

David D. Clark
Van Jacobson
John Romkey
Howard Salwen

WHILE NETWORKS, ESPECIALLY LOCAL AREA networks, have been getting faster, perceived throughput at the application has not always increased accordingly. Various performance bottlenecks 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, Transmission Control Protocol (TCP) [1]. This paper provides results of that study. Our conclusions are that TCP is in fact not the source of the overhead often observed in packet processing, and that it could support very high speeds if properly implemented.

Our conclusions are that TCP is in fact not the source of the overhead often observed in packet processing, and that it could support very high speeds if properly implemented.

TCP

TCP is the transport protocol from the Internet protocol suite. In this set of protocols, 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 Protocol (IP) [2]. This protocol, which is a connectionless or datagram packet delivery protocol, deals with host addressing and routing, but the 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 (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.

The Analysis

This study addressed the overhead of running TCP and IP (since TCP is never run without IP) and that 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 remove it from the UNIX environment, lest we be confused by some unexpected operating system overhead.

For example, the Berkeley implementation of UNIX uses a buffering scheme in which data is stored in a series of chained buffers called mbufs. We felt that this buffering scheme, as well as the scheme for managing timers and other system features,

0163-6804/89/0006-0023 $01.00 © 1989 IEEE

was a characteristic of UNIX rather than of the TCP, and that 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 altogether ignore 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 mbufs, and replaced them with working but specialized versions of the same functions. To ensure 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 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 an 80386, a Motorola 68020, or even an RISC chip such as the SPARC.

Finding the Common Path

One observation central to the efficient implementation of TCP is 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.

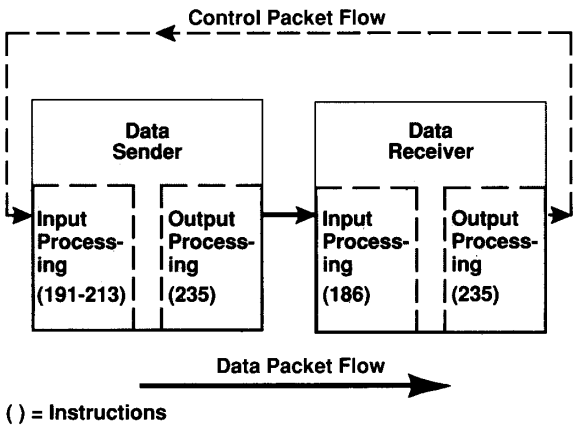

Fig. 1. Analysis terminology.

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 whether the connection was in an established state, the packet had no special control flags on, and the sequence number was expected. If so, control was transferred to the fast path. The version of the TCP that we compiled and tested had this fast path, and it was this fast path that we audited.

There are actually two common paths through the TCP, the end sending data and the end receiving it. 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 the 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. In Figure 1, we illustrate how our terms describe the various steps of the packet processing.

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. In fact, as we will discuss, this belief seems false. 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 essentially has to format the packet and start the transmission.

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

There are three general stages to the TCP processing. In the first, a search is made to find the local state information (called the Transmission Control Block, or TCB) for this TCP connection. In the second, the TCP checksum is verified. This requires computing a simple function of all the bytes in the packet. In the third stage, the packet header is processed. (These steps can be reordered for greater efficiency, as we will discuss later.)

We chose not to study the 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 and algorithms. We will return to these two operations in a later section, but in the present analysis, 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 rather different paths for the sender and receiver of data. The overall numbers are the following:

- Sender of data: 191 to 213 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 either 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 acknowledgment. Since the Berkeley TCP acknowledges at most

every other packet in a bulk data transfer, the cost in this case is 22 instructions per packet. For longer paths, the cost will be spread over more packets, so 22 instructions is an upper bound.

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 any improvement achieved would be fractional, not a gross change in overall performance. In a later section, we will return to a more global speculation on what these numbers mean.

TCP Output Processing

We subjected the output side of TCP to an analysis that was somewhat less detailed than that of the input side. We did not program a fast path, and we did not attempt to separate the paths for data sending and receiving. Thus, we 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. This number provides a rough measure of the output cost, but it is dangerous to compare it closely with the input processing cost. Neither the output side nor the input side had been carefully tuned, and both could be reduced considerably. Only if both paths had received equivalent attention would a direct comparison be justified. Our subjective conclusion in looking at the two halves of the TCP is that our starting assumption (the receiving side is more complex than the sending side) is probably wrong. In fact, TCP puts most of its complexity in the sending end of the connection. This complexity is not a part of packet sending, but a part of receiving the control information about that data in an incoming acknowledgment packet.

The Cost of IP

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

The instruction counts for IP were as follows:

- Packet receipt: 57 instructions
- Packet sending: 61 instructions

An Optimization Example—Header Prediction

The actual sending of a packet is less complex than the receiving of one. There is no question of testing for malformed packets, or of looking up a TCB. The TCB is known, as is the desired action.

One example of a simple operation is the actual generation of the outgoing packet header. 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 be likely to carry the same options.

The Berkeley implementation of UNIX makes some use of this observation, associating with each connection a template of the IP and TCP headers with a few of the fixed fields filled in. To get better performance, we designed an IP layer that created a template with all the constant fields filled in. When TCP wished to send a packet on that connection, it would call IP and pass it the template and the length of the packet. Then 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.

This idea can also be used with TCP, as was demonstrated in an earlier, very simple TCP implemented by some of us at MIT [6]. In that TCP, which was designed to support remote login, the entire state of the output side, including the unsent data, was stored as a preformatted output packet. This reduced the cost of sending a packet to a few lines of code.

A more sophisticated example of header prediction involves applying the idea to the input side. In the most recent version of TCP for Berkeley UNIX, one of us (Jacobson) and Mike Karels have added code to precompute what values should be found in the next incoming packet header for the connection. If the packets arrive in order, a few simple comparisons suffice to complete header processing. While this version of TCP was not available in time to permit us to compile and count the instructions, a superficial examination suggests that it should substantially reduce the overhead of processing compared to the version that we reviewed.

Support Functions

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 an 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 about 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, then 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 the 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 who want to deal with data one byte at a time, as well as others who want to deal in large blocks. To serve both types 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.)

The problem of the buffer layer is made worse by the fact that the protocol specifiers do not admit that such a layer exists.

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.

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 [7]. It provides very low-cost timer operations. In our version the costs were:

- Set a timer: 35 instructions
- Clear a timer: 17 instructions
- Reset a timer (clear and set together): 41 instructions

Checksums and TCBs—The Missing Steps

In the discussion of TCP input processing above, we intentionally omitted the costs 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 that is computed after the packet is actually in main memory provides a level of protection that is very valuable [8]. However, computing this checksum using the central processor rather than some outboard chip may be 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.

There are a number of processing overheads associated with processing the bytes of the packet rather than 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 96, which may not be minimal), and the number of connections that are assumed to be open.

Using the principle of the common path and caching, one can provide algorithms that are very cheap. 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 96 bits, and return the pointer. This takes very few instructions. One of us (Jacobson) added such a single-entry TCB cache to his TCP on UNIX, and measured the success rate. Obviously, for any bulk data test, where the TCP is effectively dedicated to a single connection, the success rate of this cache approaches 100%. However, for a TCP in general operation, the success rate (often called the "hit ratio") was also very high. For a workstation in general use (opening 5,715 connections over 38 days and receiving 353,238 packets), the single entry cache matched the incoming packet 93.2% of the time. For a mail server, which might be expected to have a much more diverse set of connections, the measured ratio was 89.8% (over two days, 2,044 connections, and 121,676 incoming packets).

If this optimization fails too often to be useful, the next step is to hash the 96 bits into a smaller value, perhaps an 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.

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 335 instructions. This includes the TCP and IP level processing, our crude estimate of the cost of finding the TCB and 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 an RISC instruction set. However, based on another study of packet processing code, we found little expansion of the code when converting to an 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 an RISC set.

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

At 10 MIPS, a processor can execute 400 instructions in 40 μs, or 25,000 packets/s. These processing costs permit rather high data rates.

If we assume a packet size of 4,000 bytes (which would fit in an FDDI frame, for example), then 25,000 packets/s provides 800 Mb/s. For TCP, this number must be reduced by taking into account the overhead of the acknowledgment packet. The Berkeley UNIX sends one acknowledgment for every other data packet during bulk data, so we can assume that only two out of the three packets actually carry data. This yields a throughput of 530 Mb/s.

Figuring another way, if we assume an FDDI network with 100 Mb/s 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.

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. 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 the I/O device to system address space and 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 mbuf structure, and is then moved into the user address space in a location that is aligned as the user needs it. The first copy may be done by a DMA controller or by the processor; the second is always done by the processor.

To copy data in memory requires two memory cycles, read and write. In other words, the bandwidth of the memory must be twice the achieved rate of the copy. Checksum computation has only one memory operation, since the data is being read but not written. (In this analysis, we ignore the instruction fetch operations to implement the checksum, under the assumption that they are in a processor cache.) In this implementation of TCP, receiving a packet thus requires four memory cycles per word, one for the input DMA, one for the checksum and two for the copy.[1]

A 32-bit memory with a cycle time of 250 ns, typical for dynamic RAMs today, would thus imply a memory limit of 32 Mb/s. 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.

A Direct Measure of Protocol Overhead

In an independent experiment, one of us (Jacobson) directly measured the various costs associated with running TCP on a UNIX system. The measured system was the Berkeley TCP running on a Sun-3/60 workstation, which is based on a 20-MHz 68020. The measurement technique was to use a sophisticated logic analyzer that can be controlled by special start, stop, and chaining patterns triggered whenever selected addresses in the UNIX kernel were executed. This technique permits actual measurement of path lengths in packet processing. A somewhat subjective division of these times into categories permits a loose comparison with the numbers reported above, obtained from counting instructions.

The measured overheads were divided into two groups: those that scale per byte (the user-system and network-memory copy and the checksum), and those that are per packet (system overhead, protocol processing, interrupts, and so on.) See Table I.

TABLE I. Measured Overheads

Costs*	
Per byte:	
User-system copy	200 μs
TCP checksum	185 μs
Network-memory copy	386 μs
Per packet:	
Ethernet driver	100 μs
TCP + IP + ARP protocols	100 μs
Operating system overhead	240 μs

*Idle time: 200 μs

The per-byte costs were measured for a maximum-length Ethernet packet of 1,460 data bytes. Thus, for example, the checksum essentially represents the 125-ns/byte average memory bandwidth of the Sun-3/60. The very high cost of the network-memory copy seems to represent specifics of the par-

[1]The four memory cycles per received word are artifacts of the Berkeley UNIX we examined, and not part of TCP in general. An experimental version of Berkeley UNIX being developed by one of us (Jacobson) uses at most three cycles per received word, and one cycle per word if the network interface uses on-board buffer memory rather than DMA for incoming packets.

ticular Ethernet controller chip used (the LANCE chip), which additionally locks up the memory bus during the transfer, thus stalling the processor.

The times reported above can be converted to instruction counts (the measure used earlier in the paper) using the fact (recorded by the logic analyzer) that this is essentially a 2-MIPS machine. Thus, this experiment reports 200 instructions for TCP and IP, compared to our count of about 250. This direct measurement involved a TCP with the input header prediction algorithm added, and would thus be expected to run faster than our measured version of the code. A detailed analysis of the 200 instructions seen by the logic analyzer suggests that of the total instructions, 70% were in TCP, and of those, 80% were in the output side. Thus, input header prediction seems a very effective optimization, particularly since these measurements are the total time spent per packet in a real, running UNIX system and include all the mbufs and timer processing that was simplified out of the earlier study.

The numbers here support the thesis of the last section. The per-byte operations swamp the protocol processing costs, 771 μs versus 100 μs. The protocol processing is also small compared to the operating system overhead, 240 μs versus 100 μs.

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 an acceptable cost.

By matching the memory to the special needs of packet processing, one could achieve high performance at an 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," "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 high performance.

To our knowledge, this has not been attempted. The closest example is the VMP Network Adaptor Board [9]. There are products today that provide an outboard implementation of TCP, but they seem intended more for providing ease of use and portability rather than for 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.

An alternative conclusion is that even a special outboard processor is not required. By proper design of network interface hardware, even a system like UNIX could be made to run a network protocol essentially at memory speed. The following steps seem practical, and could be undertaken in the context of UNIX today.

If the network controller were redesigned to reduce the number of interrupts per packet, the number of times the driver is executed and the actual overhead of interrupts would be reduced. If the DMA controller were removed from the network controller, and the CPU were used to move the data between device and memory, then several improvements would be achieved. While it may seem wrong to use CPU cycles to move data, note that at least one CPU copy will be required. By using the CPU to move the data from the device, the data can bypass kernel buffers altogether and flow directly from device to user memory.

A further per-byte optimization is to note that the checksum of the data can be computed in that same copy loop. This requires that the effect of the copy be ignored if the data is found to be corrupted, but this complexity is reasonable and does not affect the common path. To prove this concept, programs were carefully written by Jacobson to copy and checksum data on a 20-MHz 68020. The throughput of a checksum loop was 130 μs/kbyte, a memory-memory copy loop was 140 μs/kbyte, while a combined checksum and copy loop was only 200 μs/kbyte.

Similar experience was obtained with an 80286 at M.I.T. In NETBLT [10], 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 without further complexity until the data is moved to user memory. A version of NETBLT (for the IBM AT) was coded to combine the checksum and copy operations, and the resulting version ran considerably faster than the one with separate copy and checksum loops.

The result of combining these features is that there is now one memory cycle per byte and the data crosses the computer bus once. This is as efficient as one could expect a transfer to get. If we can assume that the overhead of the operating system can be controlled, then with a simple network interface we can imagine network performance approaching the speed of the memory of the computer.

Protocols in Silicon

It has been proposed (e.g., by the designers of XTP [11]) 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 network interface, such as a high-performance outboard processor card with special memory and controllers for byte operations such as copy and checksum, or, alternatively, to move to a fast but simple network controller with fast memory and an efficient interface to the computer bus, so that the CPU can interact with the board efficiently. In either case, a fast general-purpose processor can still be used to perform the protocol processing, using a protocol of today's generation.

If high performance is possible with a programmable element using general protocols, it is a highly desirable approach. 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 that 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 experiences.

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 suitable network interface 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. 2, pp. 2.179–2.198, Sept. 1981.

[2] Information Sciences Institute. DARPA Internet Program Protocol Specification NIC-RFC 791. DDN Protocol Handbook, vol. 2, pp. 2.99–2.149, Sept. 1981.

[3] J. B. Postel, C. A. Sunshine, D. Cohen, "The ARPA Internet Protocol," *Computer Networks*, vol. 5, no. 4, pp. 261–271, July, 1981.

[4] J. B. Postel, "Internetwork Protocol Approaches," *IEEE Transactions on Communications*, vol. Com-28, no. 4, pp. 605–611, Apr. 1980.

[5] V. Jacobson, "Congestion Avoidance and Control," *SigComm '88 Symp.*, ACM, Aug. 1988.

[6] J. H. Saltzer, D. D. Clark, J. L. Romkey, and W. C. Gramlich, "The Desktop Computer as a Network Participant," *IEEE J. on Sel. Areas in Comm.*, vol. SAC-3, no. 3, pp. 468–478, May 1985.

[7] G. Varghese and T. Lauck, "Hashed and Hierarchical Timing Wheels: Data Structures for the Efficient Implementation of a Timer Facility," *Proc. of the 11th ACM Symp. on Operating Systems Principles*, ACM Operating Systems Review, Austin, TX, Nov. 1987.

[8] J. H. Saltzer, D. P. Reed, D. D. Clark, "End-to-End Arguments in System Design," *ACM Transactions on Computer Systems*, vol. 2, no. 4, pp. 277–288, Nov. 1984.

[9] Kanakia, Hemant, and D. Cheriton, "The VMP Network Adaptor Board (NAB): High-Performance Network Communication for Multiprocessors," *SigComm '88 Symp.*, ACM, pp. 175–187, Aug. 1988.

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

[11] G. Chesson, B. Eich, V. Schryver, A. Cherenson, and A. Whaley, "XTP Protocol Definition," Technical Report Revision 3.0, Silicon Graphics, Inc., Jan. 1988.

Biography

David D. Clark received his Ph.D. from M.I.T. in 1973 and is currently a Senior Research Scientist at the Laboratory for Computer Science. His research interests include networks, protocols, operating systems, distributed systems, and computer and communications security.

After receiving his Ph.D., he worked on the early stages of the ARPAnet and managed the development of one of the first host implementations of the ARPA network protocols. Following this effort, he worked on Local Area Network (LAN) technology and was one of the developers of the token ring LAN. This effort directly led to current commercial products and was the origin of the IEEE 802.5 token ring standard.

Since the mid-'70s, Dr. Clark has been involved in the development of the Department of Defense (DOD) network protocol suite, TCP/IP or Internet. Since 1982, he has been the Chief Protocol Architect for the Defense Advanced Research Projects Agency (DARPA) in this development and currently chairs the Internet Activities Board, a steering committee sponsored by several agencies and with ties to industry, which attempts to guide the evolution of TCP/IP as it enters the commercial world. His current network research is directed towards architecture and mechanisms for the next generation of network telecommunications.

Van Jacobson studied Modern Poetry, Physics, and Mathematics at the University of Arizona and received an M.S. in physics and a B.S. in mathematics. Since 1972, he has been a Staff Scientist in the Real Time Systems Group of Lawrence Berkeley Laboratory, developing high performance, distributed, data acquisition, and control systems for particle accelerators, fusion power reactors, and rapid transit systems. He first became interested in networks in 1975 and currently devotes himself to research in networks and protocols. Since 1985, he has been an adjunct lecturer in Computer Science at the University of California, Berkeley.

John Romkey received his B.S. in computer science from M.I.T. in 1985, where he worked at the M.I.T. Laboratory for Computer Science on a number of TCP/IP-related research projects. He then founded FTP Software, a leading TCP/IP vendor for IBM PCs, in 1986, and currently does research and software development with Epilogue Technology in California.

Howard Salwen is the founder and Chairman of the Board of Proteon, Inc. Before the founding of Proteon he was Manager of System Engineering at Signatron, Inc., Lexington, MA. He held similar positions at Adcom, Inc. and at the MIT Electronic Systems Laboratory. Mr. Salwen has B.S.E.E. and M.S.E.E. degrees from M.I.T., which were received in 1958 and 1962, respectively. He is a past Chapter Chairman of the IEEE Communication Society and has organized and participated in numerous IEEE sanctioned conventions and seminars.