A Study of An Internet Protocol Implementation

David A. Mosher

Computer Science Division
University of California
Berkeley, California 94720

ABSTRACT

This report presents a detailed timing analysis of the dynamic behavior of TCP/IP and UDP/IP as they are implemented in the released 4.2BSD version of the Berkeley UNIX† operating system. The analysis is based a series of performance measurements in the kernel, directed by a specific task. The organization of the code and the algorithms used in the implementation of each routine are discussed. The effects of this organization and these algorithms on the performance of each protocol are discussed. A model is proposed to estimate the minimum cost of a protocol implementation, and a comparison is made with the measured results. The results of this paper suggest that the implementation is responsible for a good deal of the overhead in the measured performance. In addition, the overheads of both light and medium weight protocol do not appreciably differ from each other, and are overshadowed by the performance penalties due to interface to the user and to the hardware.

1. Introduction

This report presents a detailed timing analysis of the dynamic behavior of TCP/IP and UDP/IP as they are implemented in 4.2BSD. The performances of TCP, principally, and UDP have been of great concern to the Defense Department's contractors. There is a long standing debate about the use of light weight protocols, such as PUP [Boggs 80], and of medium weight protocols, such as TCP/IP [TCP 81]. Light weight protocols tend to be stateless and have simple behavior, resulting in small amounts of code with higher readability but leaving the difficult task of ensuring reliability to higher level facilities within the applications that need such reliability. Heavier weight protocols have a tendency to keep large amounts of state information and have a fairly complex state machine to drive their behavior, resulting in large volumes of hard to debug code, but freeing applications from having to deal with the problem of reliability.

In general, protocols, including TCP and UDP, are simply a functional specification of a chain of events which occur during a conversation and lead to some end result. The implementation of these functions may result in a great deal of delay (or overhead). For example, a function in which the search of a long list or table is implemented using a linear scheme may perform poorly in comparison with the same function implemented using a hashed lookup scheme. Obviously, there is a space/time trade-off here. Thus, either implementation may be more effective depending on the constraints of the architecture. Evaluating these space/time trade-offs is an important part of system design. A careful study of system usage is needed to ensure that the correct choice has been made and that the implementation behaves as was intended.

Given a specific protocol, a set of functions must be performed. It is worthwhile to consider an ideal model of how the functions should perform. This ideal performance will be referred to as

† UNIX is a trademark of AT&T Bell Labs

the functional cost of a protocol. When the protocol is implemented, the overhead may be larger. This additional costs will be referred to as the implementation cost of a protocol for a given implementation. If the implementation cost far exceeds the functional costs and we have not carefully determined both the functional cost and the implementation cost, we may be misled by simple throughput measurements of the implementations of two different protocols in an attempt to determine which protocol is more efficient.

2. The Study

To study the implementations of TCP/IP and UDP/IP provided in the 4.2BSD kernel, we exercised the kernel with user level processes, and the UNIX commands *kgmon* and *gprof* were used to provide an execution profile of the kernel. We designed two similar user level test programs, one for each protocol, which send a specific number of messages determined by their repetition count to a given host. Each test program is run with a given amount of data at a time, allowing the profile to show exactly the execution history of sending a message with that amount of data. The kernel profiling facilities were enabled only during the actual running of each test program. The test programs were run in single user mode to avoid interferences from system utilities and other user applications. (The listings of these test programs have been included in Appendix A.)

These tests were run between two VAX 11/750's‡ over a 3 megabit/second Ethernet†. One processor was used to profile the kernel while the test program was run; the other processor just sank the data from the test programs. The test programs sent a set of six different amounts of data per message: 1, 112, 113, 1023, 1024, and 1025 bytes. These amounts were chosen to study protocol behavior around boundary conditions for the network data buffer management system.

To determine the repetition count, the following calculations were made. Since the smallest packet takes 5.5 milliseconds to be shipped and the VAX 11/750 roughly executes an instruction every microsecond, this would imply that about 5,500 instructions are executed. If the profiler samples every 1/60th of a second, we would need around 91 seconds (5500 samples at 60 samples per second) of execution to allow for a hit on every instruction. Since each repetition results in at least 5.5 milliseconds of processing time, a repetition count of 17,000 would result in 91 seconds of execution time. Since our goal was to understand roughly how the time was divided and because of the limited resources available for this profiling, we compromised with a repetition count of 10,000. A greater accuracy could have been achieved with a larger repetition count, but testing time for such accuracy would be exceedingly longer.

We do not have an absolute way of judging the effectiveness, for the user applications, of the current protocols. We do have an excellent breakdown of kernel time spent while sending messages of a chosen size, and a detailed knowledge of what would happen if a user level application was to send messages of any given size.

3. Expectations

The basic function of a protocol, such as TCP or UDP, is to transfer data from one location to another. The minimal cost of such a transfer is the cost of transferring data from one location to another in memory. This cost will be referred to as the *copy cost*. The *copy cost* increases linearly as the amount of data to be transferred increases.

In order to estimate the cost of transfers from one location to another through a communications media, we need to look at the number of times the data is copied, which is an implementation issue, and the overhead per byte transferred.

In the 4.2BSD implementation, the system chooses to copy user data into system buffers and is constrained to copy the data into contiguous memory before transferring a data packet. In addition, for streaming protocols such as TCP, the data is copied into system buffers and then

‡ VAX 11/750 and VAX are trademarks of Digital Equipment Corporation

† Ethernet is a trademark of Xerox Corporation.

copied into the stream buffers. Then the data is copied from the stream buffers into data packets.

Beyond these copying costs, we have overhead due to formations of headers and the protocol state machine. These costs will vary, and represent the costs of the specific protocol. We should be able to derive these costs from the results of our study.

Thus, for UDP, we can expect a copy from user level into system buffers, and one from system buffers into data packet, plus the cost for computing the checksum of the data. For TCP, we can expect a copy from user level into system buffers, one from system buffers into stream buffers, and one from stream buffers into data packets, plus the cost for computing the checksum of the data. We include the cost of this checksum because time required for the addition of the stream of bytes for the checksum is of the same order as that for copying data from one memory location to another using a highly efficient VAX block move instruction. (The actual transfer of the data from the data packet to the medium is not measured because it happens via direct memory access by the device.)

4. How Each Protocol Works

4.1. TCP/IP

Figure 1 graphically displays the calling hierarchy of a send request for TCP/IP.

```
syscall
 write
  rwuio
   soo_rw
    sosend
     uiomove
     Copyin
     tcp_usrreq
      sbappend
      tcp_output
       m_copy
       tcp_cksum
       ip_output
        in_cksum
        enoutput
         if_wubaput
         enstart
```

Figure 1

The calling hierarchy for sending data via TCP/IP starts with a system call, *syscall*, to a generic *write* operation. *write* calls the generic user requested data routine *rwuio* which forms a user request structure. In turn, *rwuio* calls the specific routine which can perform the necessary operations for a socket, in this case *soo_rw*. *soo_rw* calls the appropriate internal routine to implement the original request to 'send data', *sosend*. *sosend* is responsible for copying data from the user level via *uiomove*, which calls *Copyin* to do the actual copying. *sosend* first determines the amount of buffer space available for this specific socket, and copies the minimum of the buffer space available and of the amount of remaining data from the user process into the system buffers. *sosend* continues to fill system buffers until all data from the user level has been queued for sending. These system buffers are then passed to the appropriate routine for handling TCP protocol requests, in this case *tcp_usrreq*. *tcp_usrreq* queues the data buffers for this TCP connection by *sbappend* and then switches immediately to *tcp_output*, the output sequencer for the TCP protocol. At this level (*tcp_output*), a specific amount of data is copied by *m_copy*. The amount of data to be sent is based on the amount of data unsent and the amount of window space available on the receiving end. This copied data will constitute the data of the stream for a TCP packet. A TCP header is formed, including a checksum by *tcp_cksum* of the header plus the data

areas. Finally the TCP packet is passed to the IP level, *ip_output*. An IP header is formed, including a checksum of the IP header computed by *in_cksum*. Finally the message is queued for the specific network interface for transmission. In this implementation, the network interface consists of *en_output* and *en_start*. Before such a transmission can happen over this network interface, the buffered header and data must be located in a single contiguous memory space; this is done by *if_wubaput*.

4.2. UDP/IP

Figure 2 graphically displays the calling hierarchy of a send request via UDP/IP.

```
syscall
 sendto
  sendit
   sosend
    uiomove
     Copyin
    udp_usrreq
     udp_output
      udp_cksum
      ip_output
       en_output
        if_wubaput
        en_start
```

>From the user process' viewpoint, sending data through UDP/IP is done by issuing a system call equivalent to *write*, which is *sendto*. Each call of *sendto* requires the destination address. *sendit* is called, which in turn calls *sosend* as TCP/IP does. Again *sosend* calls *uiomove*, which calls *Copyin* to actually copy the user data into system buffers. These buffers are given to *udp_usrreq* which calls *udp_output* after a pseudo connection is established. As with TCP, *udp_output* represents the output sequencing of the UDP protocol. At this level(*udp_output*), the UDP header is formed. The header and data are checksummed via *udp_cksum*, and then passed to *ip_output* as with TCP/IP. *ip_output* forms the IP header, checksums the header via *in_cksum*, and passes the buffered header and data to the appropriate network interface, in this case *en_output* and *en_start*. Again, the buffers must be located in a single contiguous memory space before transmission; this is done in *if_wubaput*.

>From our discussion in the previous section, *uiomove* and *Copyin* represent the copying from user to system space and *if_wubaput* represents the copying from buffers to a single contiguous packet. For TCP/IP, *m_copy* represents the additional copying from system buffers to stream buffers. The costs of *tcp_cksum* and *udp_cksum* are of the same order as the above copy costs.

5. The Results

Using the test programs and the profiling facilities, we were able to obtain in seconds the processing time of each routine called by our test program for a message with a specific amount of data. These timings are shown in the following tables, Table 1 and 2. The values in the tables were obtained by dividing appropriately what was observed by the repetition count, and represent the number of milliseconds spent in each routine.

Routine	Number of bytes sent by User					
	1	112	113	1023	1024	1025
syscall	.35	.30	.34	.33	.28	.37
sendto	.08	.09	.09	.08	.09	.08
sendit	.29	.28	.29	.28	.27	.25
sosend	**.50**	**.57**	**.67**	**1.92**	**.56**	**.74**
uiomove	**.12**	**.15**	**.20**	**1.07**	**.13**	**.25**
Copyin	**.11**	**.20**	**.24**	**1.46**	**.85**	**.85**
udp_usrreq	.18	.19	.15	.20	.16	.19
udp_output	.26	.25	.23	.36	.30	.28
udp_cksum	**.22**	**.34**	**.24**	**1.28**	**.88**	**.83**
ip_output	.41	.39	.35	.43	.40	.41
ip_cksum	.42	.45	.44	.34	.36	.40
enoutput	.34	.31	.30	.35	.38	.29
if_wubaput	**.40**	**.43**	**.50**	**1.49**	**.40**	**1.05**

Table 1
UDP Kernel Profiling per Routine

Routine	Number of bytes sent by User					
	1	112	113	1023	1024	1025
syscall	.29	.28	.28	.30	.30	.31
write	.07	.08	.08	.08	.08	.09
rwuio	.16	.17	.18	.18	.18	.22
soo_rw	.09	.08	.08	.08	.08	.08
sosend	**.39**	**.48**	**.62**	**1.55**	**.55**	**1.65**
uiomove	**.09**	**.10**	**.17**	**.92**	**.14**	**.88**
Copyin	**.04**	**.11**	**.17**	**1.10**	**.69**	**1.12**
tcp_usrreq	.22	.22	.19	.20	.20	.22
sbappend	**.16**	**.16**	**.21**	**.83**	**.11**	**.80**
m_copy	**.28**	**.39**	**.52**	**1.82**	**.24**	**2.97**
tcp_output	.63	.62	.63	.85	.71	**1.12**
tcp_cksum	**.22**	**.31**	**.36**	**1.62**	**1.05**	**1.66**
ip_output	.28	.30	.31	.34	.34	.56
in_cksum	.19	.17	.16	.37	.26	.44
enoutput	.29	.34	.34	.31	.41	.50
if_wubaput	**.38**	**.40**	**.53**	**1.42**	**.47**	**1.66**

Table 2
TCP Kernel Profiling per Routine

These values are only intended to show the relative ordering of the routines with respect to time consumption, and to show gross changes in the magnitude of the time consumed by each routine. Since the same amount of data is copied for both TCP and UDP for at least the 1 and 112 byte case, we can see that these timings results vary significantly. Absolute timings require careful attention to the stability of the results and sufficient samples to provide stable results. In these tables, we have highlighted in boldface those routines which are most sensitive to variations of the amount of data to be processed.

6. Analysis

For both TCP and UDP, a user process request to send data must be processed by the socket scheme of 4.2BSD and finally passed to the IP layer for transport on the network. Thus, we can try to factor those costs common to both TCP and UDP to understand their global costs. The following sections analyze the results in Tables 1 and 2.

6.1. TCP/IP

>From Table 2, we can see that, of the 16 different routines used in TCP/IP, only 8 present processing costs which vary significantly with the amount of data sent. The processing time of the other 8 routines remains practically constant.

These timings are relatively constant until we reach the socket processing routines, specifically *sosend*. *Sosend*, called once indirectly from the user level, generates a chain of system buffers which contain the user level data. These buffers are referred to as memory management buffers (mbuf's), defined by the *mbuf* structure. These buffers are used either to hold data or to reference data. When the buffer is used to hold up to 112 bytes, it is referred to as a small mbuf. When the buffer is used to reference data, it can reference a page of data, 1024 bytes, and is referred to as a large mbuf. All lower routines in the calling hierarchy must follow this mbuf chain and process the data in each mbuf. *uiomove* gets called to fill each mbuf and does no processing of the data. *Copyin* is also called once per mbuf and copies the data from the user process to the data portion of the mbuf. The amount of data copied has a fixed linear cost, but the procedure call and setup cost is significant. For data transfers of less than a page, i.e. less than 1024 bytes, the data is copied into a chain of small mbuf's. Thus the difference between the overhead of copying 1023 bytes and that of copying 1024 bytes is almost entirely due to the number of mbuf's or the number of times *Copyin* is called. *sbappend* puts these mbuf's on a queue for this TCP connection and tries to combine the data of two mbuf's into one. *sbappend* is affected by the length of the mbuf chain and the utilization of the data portion of each mbuf. The same is true for *m_copy*, except that, in the case of a large mbuf, the data is not copied but referenced. This results in a dramatic savings when large mbuf's are used. The time spent in *tcp_cksum* is affected by the number of mbuf's and by the amount of data in each mbuf as well. The checksum is computed for blocks of data. Each mbuf causes a premature disruption of this block processing of the checksum and this results in substantial overhead. *if_wubaput* is very similar to *Copyin* except that it copies data from mbuf's to a contiguous segment. In the case of 1024 bytes, the data is contiguous and need not be copied. If only a portion of a large mbuf is being sent, the whole referenced data of the mbuf must be recopied into a page aligned segment. It would appear that the buffer management strategy has a significant impact on the performance of this implementation. In the case of 1024 bytes, we see that a different buffer management strategy, using only large mbuf's, greatly reduces the overhead, and checksumming becomes the most expensive task.

6.2. UDP/IP

In Table 1 we can see that, of the 12 different routines used in UDP/IP, only 5 present processing costs which vary significantly with the amount of data sent. The processing times of the other 7 routines remain practically constant. As with TCP, *sosend* is responsible for the generation of the mbuf chain which will directly affect the processing at the lower levels. As with TCP, *uiomove* is called once per mbuf, and substantial overhead results from processing the mbuf chains. *Copyin* suffers from the overhead of processing the mbuf chain, but its processing time does vary as the amount of data to be copied varies. We can see from the difference between 1023 bytes and 1024 bytes that mbuf chaining results in a great deal of overhead which can not be accounted for by the copying of a single extra byte. *udp_cksum* is similarly affected by the mbuf chain. As with TCP, the copying of the data from the mbuf chain into a single contiguous segment is dramatically affected by the number of mbuf's. Note the dramatic change for 1024 bytes in *if_wubaput*, where the data is not copied but referenced. Note that, for 1025 bytes, the time taken in *if_wubaput* jumps higher, but not as high as for 1023 bytes. TCP has additional processing related to its acknowledgements and window updates, which are not provided for in UDP. If the user wished to guarantee the arrival of all of the data, an UDP acknowledgement scheme would be needed, resulting in system overhead to send the acknowledgement packet from receiving user process to the sending user process. As with TCP for the case of 1024 bytes, the most expensive remaining cost is that of checksumming. Clearly the buffering scheme chosen in *sosend* has a dramatic effect on the processing at the lower levels of both protocol implementations.

We can see from Table 1 and Table 2 that UDP and TCP have some of the same critical routines, all of which are sensitive to mbuf chains. We can also see that UDP has little more to optimize than these routines, so improvements beyond those that might result from the reduction of the mbuf chains seem unlikely. TCP has additional critical routines which may be improved by better buffer management but which would still have a sizeable impact on the performance for this implementation of TCP. >From the case of 1024 bytes were the costs of mbuf chaining is minimal, we can see that the costs of the state sequencing routines in UDP and TCP, *udp_output*, and *tcp_output*, vary by a factor of 2 but have only a minor impact on the total performance.

6.3. Comparison of TCP and UDP

We can view the activities of each protocol's implementation in a different way by looking at the number of times each routine was called. Table 3 presents, for TCP/IP, such a decomposition, while Table 4 has it for UDP/IP.

	Number of bytes sent by user process					
Routine	1	112	113	1023	1024	1025
syscall	10,882	10,884	10,882	10,884	10,882	10,882
write	10,002	10,002	10,002	10,002	10,002	10,002
rwuio	10,009	10,010	10,009	10,010	10,009	10,009
soo_rw	10,000	10,000	10,000	10,000	10,000	10,000
sosend	10,000	10,000	10,000	10,000	10,000	10,000
uiomove	**10,014**	**10,015**	**20,014**	**100,015**	**10,014**	**97,158**
tcp_usrreq	10,011	10,020	10,021	10,119	10,022	10,046
sbappend	10,000	10,000	10,000	10,088	10,000	10,000
tcp_output	10,010	10,017	10,015	10,104	10,008	10,015
m_copy	10,007	10,022	10,024	10,136	10,010	**20,022**
tcp_cksum	*10,333*	*11,497*	*11,671*	*20,284*	*20,256*	*30,087*
ip_output	10,016	10,049	10,032	10,182	10,024	**20,081**
in_cksum	*10,384*	*11,628*	*11,780*	*20,494*	*20,356*	*30,359*
enoutput	10,016	10,049	10,032	10,182	10,024	**20,081**
if_wubaput	10,016	10,049	10,032	10,182	10,024	**20,056**
enstart	10,024	10,059	10,037	10,188	10,057	**20,141**
ipintr	356	1,532	1,686	10,035	10,238	10,012
tcp_input	323	1,472	1,645	10,095	10,243	10,012

Table 3
Number of Calls per Routine for 10,000 TCP Send Requests

First we note that *sosend* is called by each protocol exactly the same number of times. In addition, *uiomove* and the uncounted *Copyin*†, *ip_output*, *enoutput*, and *if_wubaput* are called about the same number of times for data amounts of 1 byte through 1024 bytes. Thus, the behavior of these two protocols is approximately the same under these conditions for this implementation. Since both protocol implementations called *ip_output* about the same number of times, we can deduce that no buffering occurs from system call to system call. Such buffering is permissible for TCP, but is not for UDP. As noted before, TCP requires some additional processing due to its retransmission concerns, represented by *m_copy*, *ipintr*, and *tcp_input*. As well, calls to *ipintr* and *tcp_input* result in a proportionally higher number of calls to *in_cksum* and *tcp_cksum*.

We see quite a dramatic difference when the packet size reaches 1025 bytes of data, between UDP and TCP as implemented in 4.2BSD. We note for UDP that *ip_output* is still only called

† *Copyin* is an assembly language routine and cannot be instrumented with profiling from the C compiler

Routine	Number of bytes sent by user process					
	1	112	113	1023	1024	1025
syscall	10,886	10,888	10,886	10,888	10,886	10,886
sendto	10,000	10,000	10,000	10,000	10,000	10,000
sendit	10,000	10,000	10,000	10,000	10,000	10,000
sosend	10,000	10,000	10,000	10,000	10,000	10,000
uiomove	**10,015**	**10,016**	**20,015**	**100,016**	**10,015**	**20,015**
udp_usrreq	10,002	10,002	10,002	10,002	10,002	10,002
udp_output	10,000	10,000	10,000	10,000	10,000	10,000
udp_cksum	10,000	10,000	10,000	10,000	10,000	10,000
ip_output	10,019	10,017	10,025	10,036	10,025	10,057
in_cksum	*29,021*	*28,820*	*28,458*	*28,710*	*29,818*	*29,770*
enoutput	10,019	10,017	10.025	10,036	10,025	10,037
if_wubaput	10,019	10,017	10,025	10,036	10,025	10,037
enstart	10,037	10,039	10,037	10,056	10,028	10,044

Table 4

Number of Call per Routine for 10,000 UDP/IP Send Requests

once per *sosend*, while for TCP *ip_output* is called twice per sosend. The second call is due to the fact that the maximum segment size is 1024 bytes, so that the data must be sent in two packets. This additional call to *ip_output* results in additional calls to lower level routines such as *tcp_cksum*, *in_cksum*, *m_copy*, *enoutput*, *enstart*, and *if_wubaput*. In addition, we see a considerable difference between the number of times *uiomove* and the uncounted *Copyin* are called for 1025 bytes between UDP and TCP. This difference is found at the socket level and would appear not to be related to the protocol.

This behavior of the TCP implementation for packet sizes greater than 1025 bytes needs to be careful studied and explained. We can clearly see that the number of mbuf's used must be significantly higher for TCP than for UDP since the number of times *uiomove* is called is related to the number of mbuf's created. We would expect the number of calls to be one for 1024 bytes for a large mbuf, and one for 1 byte for a small mbuf (as seen with UDP).

For TCP, the socket level scheme in *sosend* is dependent not only upon the amount of data to be spent but also upon the amount of space available in the buffer. UDP does not use socket buffering, but is limited to sending packets no larger than 2048 bytes. The buffer space for sockets was 2048 bytes in TCP. So, if we send 1025 bytes on a TCP socket, buffered as 1024 bytes in a large mbuf and 1 byte in a small mbuf, only 1023 bytes remain available in the buffer. When a second send of 1025 bytes occurs before the previous 1025 bytes have been acknowledged, only 1023 bytes may be copied into the buffer. Since 1023 bytes is smaller than a large mbuf, the data must be placed in small mbuf's. When the last packet of 1025 bytes is acknowledged, the remaining 2 bytes may be copied in, leaving again 1023 bytes available in the buffer space. But since the 2 bytes where just sent, our next acknowledgement would be for the 1023 byte packet, leaving only 1021 bytes available in the buffer. Depending on the amount of data acknowledged per acknowledgment packet, a very complex behavior similar to the silly window syndrome[Clark 82] found at the TCP level occurs here at the socket level. As we noted previously, TCP will send two packets for each 1025 bytes, one 1024 bytes and one 1 byte. Because of the delayed acknowledgement scheme of this implementation, the acknowledgement is for 1025 bytes. We can see that the number of acknowledgements happens to be about the same as those of the users system calls, *ipintr* and *sosend*, respectively. We would then expect the number of calls to *uiomove* to be about the same as for the 1023 byte case. The number of calls to *uiomove* for 1025 bytes is slightly less, which indicates that sometimes the acknowledgement must have happened before the user process had a chance to send its next 1025 bytes of data. This behavior is controlled by the size of the buffer and the amount of delay in the acknowledgement. We can see from the 1024 byte case that, for every 1024 bytes sent, the acknowledgement appears to occur before the next 1024 bytes can be sent and acknowledged. If acknowledgements were sent much later, we might

see a more regular behavior. This behavior deserves more careful study in the future.

7. Conclusions

This report has presented a detailed timing analysis of the dynamic behavior of TCP/IP and UDP/IP as they are implemented in 4.2BSD. We have discussed the importance of accessing the functional cost and the implementation cost so as not to be misled by simple measurements of a particular protocol's implementation. We have employed user level programs to exercise the kernel and studied protocol behavior with the given memory management scheme. We have discussed the system behavior for both stream and datagram protocols, including costs in terms of the number of times the data was copied. We have given a detailed description of the processing of both TCP and UDP send data requests for each network layer. We have discussed the limitations of the operating system interface and of the hardware interface. Measurements of the time spent in each routine have been presented to show time consumptions and gross changes in the magnitude of such consumptions. An analysis of these measurements was given. For both protocols, the buffer scheme used in the implementations appears to have a substantial impact on performance. In the case of 1024 bytes, checksumming becomes the most expensive task. When further analyzing the number of times each routine is called, we have seen similar behaviors for TCP and UDP for data transfers of less than one page. For transfers of more than one page of data, we have discovered an unexpected behavior at the socket level, which dramatically affects the management of the data buffers for stream protocols, such as TCP. Overhead is also added by the constraints of the maximum segment size limitation within TCP. We have seen that the routines which implement the state sequencing for TCP and UDP differ by a factor of two in processing time, but that this difference is small in comparison with the total costs of sending data.

8. References

[Boggs
David R. Boggs et al., "Pup: An Internetwork Architecture." IEEE Transactions on Communications, Vol. COM-28, No. 4, April 1980, pp. 612-624.

[TCP
"Transmission Control Protocol." RFC 793, Information Sciences Institute, Marina del Rey, California, September 1981.

[Clark
Dave Clark, "Window and Acknowledgement Strategy." RFC 813, Information Sciences Institute, Marina del Rey, California, July 1982.