The Impact of Buffer Management on Networking Software Performance in Berkeley UNIX 4.2BSD: A Case Study.

Luis Felipe Cabrera
Michael J. Karels
David Mosher

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA

INDEX

The Impact of Buffer Management on Networking Software Performance in Berkeley UNIX† 4.2BSD: A Case Study.

Luis Felipe Cabrera ‡
Michael J. Karels
David Mosher

Computer Systems Research Group
Computer Science Division
Department of Electrical Engineering and Computer Sciences
University of California, Berkeley
Berkeley, CA 94720, USA

Abstract

Berkeley UNIX 4.2BSD is an operating system which provides easy networking among 4.2BSD installations and others supporting DOD's Internet protocols. Moreover, it also offers alternative ways for processes to communicate with each other both within and across machine boundaries. Processes need not have a common ancestor to communicate and they may do so using different addressing families and styles of communication. In addition, several protocol families may be supported simultaneously.

In this paper we present a detailed timing analysis of the dynamic behavior of the TCP/IP and the UDP/IP network communication protocols' current implementation in Berkeley UNIX 4.2BSD. These measurements show the effect that kernel buffer management has on the network software performance. We discuss issues and tradeoffs involved when implementing network communication mechanisms for multiple-protocol systems. We highlight the intricate interrelationships arising from the simultaneous coexistence of different buffering policies within a system.

This study also sheds light on the inefficiencies encountered when software and hardware perform the same actions on data, e.g., checksums.

Index Terms: Berkeley UNIX, 4.2BSD, benchmarking, interprocess communication, datagram, virtual circuit, TCP protocol, UDP protocol, IP protocol, Ethernet, artificial workload, dynamic program profile.

1. Introduction

Berkeley UNIX 4.2BSD is an operating system which provides alternative ways for user processes to communicate with each other [9-10, 19, 23]. User processes may choose intermachine communication medium, protocols [15-17], addressing families, and styles of communication. In particular, user processes may use datagram or stream communication. In Berkeley UNIX 4.2BSD two processes wishing to communicate need not have a common ancestor nor reside in the same host. In this paper we present a study of the dynamic behavior of TCP/IP and UDP/IP in which the impact that kernel buffering has on network software performance is highlighted. We have studied the implementations which existed at Berkeley during the summer of 1984, nine months

† UNIX is a Trademark of AT&T Bell Laboratories

‡ On leave from the Departamento de Ciencia de la Computación of the Escuela de Ingeniería of the Pontificia Universidad Católica de Chile.

This work was sponsored by the Defense Advanced Research Projects Agency (DoD), monitored by the Naval Electronics Systems Command under contract No. N00039-84-C-0089. The views and conclusions contained in this document are those of the authors and should not be interpreted as representing official policies, either expressed or implied, of the Defense Research Projects Agency or of the US Goverment.

after 4.2BSD release, in Ethernet based environments [13-14]. Hereafter, we shall refer to this version of Berkeley UNIX 4.2BSD as the 'current' system. (The system we measured already differed in many aspects with the released 4.2BSD Berkeley UNIX.)

A secondary aspect we wanted to observe was the possible effect that the reliability built into the TCP protocol [17, 14] has on the user process-perceived network latency and overall network performance. Two questions we are concerned with are: What are the tradeoffs of using TCP compared to UDP if the underlying physical network does not lose packets? Given that some Ethernet interface hardware provides CRC checksums, what is this cost for user applications when using TCP?

The rest of this paper is subdivided as follows. In Section 2 we present the basic measurement assumptions. Section 3 has a discussion of the experimental environment used in this study. Section 4 contains a detailed dynamic timing analysis of the current TCP/IP and UDP/IP implementations. Finally, Section 5 consists of our conclusions.

2. Timing From User Process Space

Even though many studies have evaluated and modelled the performance of Ethernets under various conditions [1-4, 6-7, 18, 20-21], measurements performed in most of those studies have only determined the extreme limitations of this technology, and the degrees of performance degradation which can be expected at the lowest level, i.e., when analyzing communication performance from the hardware interfaces viewpoint. Indeed only [2, 7, 21-22] have measurements of the performance a software system may expect. However, in [21-22] those measurements do not refer to communications between user processes but between operating system processes. In [8] we find an analysis which includes modelling the behavior of Ethernets under different parametric load conditions, in the context of file servers.

In this paper we study the network software performance perceived by user processes. It should be emphasized that all measurements were made from user space, and thus they include all the overhead caused by the protocol implementations and by the operating system. Better understanding of how our local area network implementations perform makes it easier to design distributed applications. It also provides a better knowledge of the current limitations of these distributed applications, and where the improvements may come from.

3. The Experimental Environment

This paper describes a series of tests, performed at UC Berkeley, designed to determine the dynamic performance properties of the TCP/IP and UDP/IP Ethernet-based IPC mechanisms under Berkeley UNIX 4.2BSD. Given that IP, as currently implemented, cannot be accessed directly by user processes (but can by privileged processes), we shall not report explicitly about it in this paper. It should be clear, however, that since UDP and TCP use IP, the performance of its implementation affects user process applications.

For our study we used the kernel configured for profiling and executed ad hoc routines to stress desired aspects of the network subsystem. The assessment was then done through the use of the commands *kgmon* and *gprof* [5, 11]. *gprof* profiling is known to add 15 to 20% cpu overhead to the profiled code [11]. This should be kept in mind while reading the timings in Section 4. A test program which sends a fixed amount of data a predetermined number of times was designed for each protocol. The kernel monitoring facility was enabled during the execution of the test program. All tests were run in single user mode to avoid interferences. The profiles were determined later from the data collected by the monitor [5].

3.1. The Sizes of Messages and the Repetition Count

The six user message sizes used for the network tests were: 1, 112, 113, 1023, 1024, and 1025 bytes. This set of message sizes was chosen to stress border conditions in the buffer management schemes provided by the protocol implementations. Some of them also represent traffic resulting

from common network operations, as is the case with the 112 byte size and the 1024 byte size. Most system utilities use, for example, the 1024 byte size as it represents one logical page of data in Berkeley UNIX 4.2BSD. Indeed the networking software has been optimized for the 1024 byte size.

The networking software manages its own address space with two sizes of buffers: 128 bytes and 1024 bytes. Mbufs are 128 byte buffers where 8 bytes are used for link pointers, 4 bytes for the data offset, 2 bytes for the size, and 2 bytes for the type. Mbufs may contain up to 112 bytes of data, or they may point to an associated 1024 byte data area which always contains exactly 1024 bytes of data. User supplied data is copied into a chain of mbufs with 1024 byte buffers for each segment that is at least a page in size, and any remainder (or all if the segment is less than 1024 bytes) is copied into mbufs containing up to 112 bytes each. Internal copy operations involve physical copying of the data segments inside of mbufs. Mbufs which point to 1024 bytes of data are copied by augmenting an associated reference count. Internal copy operations are required by the retransmission algorithm with TCP and for messages to be assembled into one contiguous buffer after being given to the network driver. In Section 4 the effect that saving one of these copy operations has on network performance will become apparent. IP protocol message fragmentation was not assessed in this study.

It should be remarked that, for every message to be sent, the network software always prepends a 128 byte mbuf for the 40 byte header that is required by the TCP/IP and UDP/IP protocols. Messages with 1024 byte data segments may be transmitted by the link layer using a trailer protocol [9, 12], which allows the TCP/IP or UDP/IP header to be moved to the end of the data area in order to page-align the data area for both the transmitting and receiving hosts. This optimization, which avoids message reassembly costs at the receiver end of a transmission, can only be done while the headers of messages have constant length.

Each network test consisted of a user process sending a fixed size message a predetermined number of times. These tests were run between two VAX 11/750's over a private 3 megabit/second ether. One processor was used to profile the kernel while the test program ran; the other otherwise idle processor just sinks the data from the test programs. We call 'repetition count' the number of times each packet was sent. The higher the repetition count, the larger the degree of accuracy we could obtain from our profiling tools. To obtain values accurate to one millisecond a repetition count of 10,000 was necessary. The same repetition count was used in each of our runs.

4. Assessment of Protocol Implementation

In this section we present the study of the implementations of TCP/IP and UDP/IP in detail. The primary goal of this study is to understand the specific costs of using TCP/IP and UDP/IP as currently implemented. Secondarily, this study has pointed out unexpected performance penalties for certain message sizes.

Two similar test programs were designed, one for each protocol, which sent a specified number of messages of a specified length to a predetermined host. In each run, from the profiled kernel we obtained the execution history of sending a message with the specified 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 as much as possible. The test programs have been included in Appendix A.

It must be remarked, however, that lacking a hardware monitor to measure overall ether message size distribution we did not have an absolute way of judging the effectiveness of the current protocol implementations for the user process applications in general. What we do have, however, is an excellent breakdown of kernel time spent while sending messages of the chosen sizes, and a detailed knowledge of what would happen if, say, a user space application sent messages of any given size.

The raw data from these profiles appears in the following three subsections. The values in Tables 1 and 2 represent the number of seconds spent processing in each routine during the 10,000 transmissions. Because these values were obtained from a single run for each message size, they are mostly intended to show the relative ordering of the routines with respect to processor utilization, and to show gross changes in the magnitude of time utilization of each routine. Obtaining absolute timings would require further data stability analysis. (Some of the message sizes were run more than once on different days. No significant timing differences were observed.) In Tables 1 and 2 we have highlighted in boldface those routines which exhibit larger timing variations as a function of the amount of data to be processed. We have not tried to factor out the 'Heisenberg' effect of *gprof*'s overhead.

For both protocols, the buffer scheme used in the implementation appears to have an overwhelming effect on performance. Since UDP/IP sends data atomically, and only limits a message's maximum size, this protocol is not so sensitive to varying data sizes. The drastic increases in overhead in the routines shown in bold appear to be due to the data buffer management scheme chosen. On the other hand, TCP/IP, with its windows and data streaming, is sensitive to the amount of data presented. Thus, in addition to the increased overhead seen in the routines which deal with data within the buffer management system, the actual protocol implementation overhead appears to increase noticeably because of the varying amounts of data presented.

A word of caution. Our analysis of TCP/IP was done based on a user process which sent data using *write*, while that of UDP/IP used *sendto*. These calls have, for example, different number of parameters and thus their overheads have to be compared with care.

4.1. TCP/IP

Table 1 presents the time spent in a selected group of routines that are called to process a TCP/IP transmission with a specific amount of data. These values were obtained directly from the *gprof* output.

The calling hierarchy for sending data via TCP/IP starts with a system call, *syscall*, to a generic *write* operation. *write* calls *rwuio* to set up transfer data structures. In turn, *rwuio* calls the specific routine which can perform the necessary operation for the type of object, in this case *soo_rw*. *soo_rw* calls the appropriate internal routine that implements the original request to 'send data', *sosend*. *sosend* is responsible for allocating buffers and copying the data from the user space via *uiomove*, which calls *Copyin* to do the actual copying. *sosend* first determines the amount of buffer space available for this specific socket, and then copies the minimum of the buffer space available or the amount of data to be sent, whichever is smaller, into mbufs. These mbuffs are then passed to the appropriate protocol, in this case *tcp_usrreq*. *tcp_usrreq* queues the data buffers for this TCP connection with *sbappend* and then switches immediately to *tcp_output*, the output sequencer for the TCP protocol. Based on the windowing policies, an amount of data to be sent is selected. This data is copied from the TCP output queue by *m_copy*. Data and header are checksummed in *tcp_cksum*, and then passed to the IP level, *ip_output*. The additional IP header information is checksummed in *in_cksum*. Finally, the message is queued and possibly sent to the specific network interface for transmission. In this study, the network interface is represented by the two functions *en_output* and *en_start*. Before such a transmission can happen, the buffered data must be copied and mapped into a single contiguous memory space; this is done in *if_wubaput*.

From the row entries in Table 1, we can see that, of the 21 different routines listed for TCP/IP, 12 present processing costs which vary significantly with the amount of data sent. The processing time of the other 9 routines remains practically constant. The five calls which show a larger variation in the vicinity of the 1024 bytes region are *sosend*, *uiomove*, *m_copy*, *sbappend*, and *if_wubaput*. All are associated with buffer management. The four calls which have a larger impact in the processing of messages are *sosend*, *if_wubaput*, *tcp_cksum*, and *m_copy*.

TCP/IP Routines and System Calls	Message Size in Bytes					
	1	112	113	1023	1024	1025
syscall	2.91	2.79	2.76	3.02	3.01	3.12
write	0.72	0.81	0.81	0.78	0.84	0.90
rwuio	1.57	1.67	1.82	1.77	1.84	2.23
soo_rw	0.92	0.83	0.85	0.82	0.77	0.84
sosend	**3.89**	**4.76**	**6.24**	**15.53**	**5.53**	**16.52**
uiomove	**0.99**	**1.02**	**1.77**	**9.23**	**1.38**	**8.85**
Copyin	**0.45**	**1.17**	**1.68**	**10.96**	**6.86**	**11.19**
ipintr	**0.19**	**0.93**	**0.94**	**6.31**	**4.88**	**5.11**
tcp_usrreq	2.20	2.20	1.93	2.00	2.03	2.24
tcp_input	**0.06**	**1.32**	**1.67**	**11.14**	**10.68**	**10.97**
tcp_output	**6.26**	**6.16**	**6.34**	**8.50**	**7.14**	**11.23**
tcp_xoutput	**0.00**	**0.38**	**0.28**	**2.11**	**1.68**	**5.45**
sbappend	**1.65**	**1.63**	**2.11**	**8.33**	**1.11**	**8.02**
ip_output	2.78	3.07	3.14	3.36	3.35	5.63
tcp_cksum	**2.23**	**3.13**	**3.65**	**16.16**	**10.52**	**16.72**
in_cksum	**1.89**	**1.72**	**1.57**	**3.69**	**2.66**	**4.41**
m_copy	**2.77**	**3.86**	**5.22**	**18.24**	**2.38**	**29.73**
enoutput	2.74	3.38	3.45	3.09	4.15	5.08
enstart	2.87	2.53	2.81	2.17	2.78	4.53
if_wubaput	**3.83**	**4.00**	**5.30**	**14.23**	**4.70**	**16.58**
in_lnaof	2.44	2.21	2.23	2.76	2.72	3.43
total of boldface routines	**24.21**	**30.08**	**36.77**	**124.43**	**59.52**	**144.78**
total of lightface routines	19.15	19.49	19.80	19.77	21.49	28.00

Table 1: Partial Decomposition of TCP/IP Processing Time in Seconds for 10,000 Transmissions Between two Dedicated VAX 11/750. [*Highlighted in boldface are those calls with larger timing changes.*]

gprof of TCP/IP	Message Size in Bytes					
	1	112	113	1023	1024	1025
Number of routines	233	266	260	264	265	265

Table 1a: Number of Routines in the Kernel **gprof** Profiling for TCP/IP.

tcp_xoutput, which is a copy of *tcp_output*, is only called from *tcp_input*. This allowed us to observe in isolation the cost of the flow control mechanism and of packet acknowledgement. Clearly, the greatest impact comes from those routines which do copying of data within the interfaces. In the 1024 case, checksumming and servicing acknowledgements and window updates through *tcp_input* and *tcp_xoutput* are the most expensive tasks.

As mentioned in Section 1, there are network hardware interfaces which provide checksumming facilities. As can be observed from the entries for *in_cksum* and *tcp_cksum*, the time spent in it is substantial. This, in fact, is true for both protocols (see Tables 1 and 2). It is clear, then, that redundant checksumming in a system has definite performance penalties.

4.2. UDP/IP

Table 2 presents the time spent in a selected group of routines that are called to process a UDP/IP datagram with a specific amount of data. These values were obtained directly from the *gprof* output.

UDP/IP Routines and System Calls	Message Size in Bytes					
	1	112	113	1023	1024	1025
syscall	3.48	3.02	3.40	3.29	2.76	3.67
sendto	0.84	0.94	0.91	0.83	0.99	0.85
sockargs	0.80	0.93	0.87	0.81	0.77	0.84
getsock	0.79	0.62	0.46	0.63	0.64	0.71
sendit	2.92	2.84	2.94	2.85	2.69	2.46
m_freem	**2.59**	**2.35**	**2.98**	**3.20**	**4.43**	**2.28**
useracc	0.56	0.55	0.55	0.60	0.63	1.03
sosend	**5.07**	**5.75**	**6.68**	**19.25**	**5.58**	**7.38**
uiomove	**1.20**	**1.48**	**2.05**	**10.66**	**1.34**	**2.52**
Copyin	**1.14**	**2.02**	**2.37**	**14.58**	**8.45**	**8.47**
udp_usrreq	1.83	1.90	1.52	2.00	1.56	1.91
in_pcbconnect	2.24	2.45	2.13	2.56	2.65	2.17
in_pcblookup	0.95	1.13	0.81	1.28	0.96	1.03
in_netof	2.67	2.42	3.16	3.27	3.13	3.33
if_ifonnetof	0.46	0.46	0.53	0.64	0.73	0.62
in_pcbdisconnect	0.52	0.64	0.53	0.55	0.55	0.46
udp_output	2.61	2.45	2.28	3.66	3.07	2.78
m_get	**1.96**	**1.46**	**1.86**	**1.93**	**3.54**	**2.06**
ip_output	4.12	3.89	3.53	4.26	4.01	4.14
udp_cksum	**2.23**	**3.37**	**2.40**	**12.82**	**8.79**	**8.28**
in_cksum	4.19	4.54	4.43	3.44	3.56	4.01
in_lnaof	2.25	2.41	2.43	2.44	2.40	2.27
ipintr	4.42	3.71	3.96	2.69	4.34	4.36
enrint	3.07	3.51	4.44	3.37	3.90	4.14
enoutput	3.36	3.12	3.04	3.46	3.87	2.80
enstart	3.16	2.88	2.50	2.66	2.02	2.67
if_wubaput	**4.02**	**4.31**	**5.08**	**14.96**	**4.01**	**10.54**
getf	0.39	0.36	0.41	0.27	0.48	0.44
total of boldface routines	**18.21**	**20.74**	**23.42**	**77.40**	**36.14**	**41.53**
total of lightface routines	45.63	44.77	44.83	45.56	45.71	46.69

Table 2: Partial Decomposition of UDP/IP Processing Time in Seconds for Sending 10,000 Datagrams Between two Dedicated VAX 11/750. [*Highlighted in boldface are those calls with larger timing changes.*]

gprof of UDP/IP	Message Size in Bytes					
	1	112	113	1023	1024	1025
Number of routines	261	252	261	254	256	255

Table 2a: Number of Routines in the Kernel **gprof** Profiling for UDP/IP.

From the user process viewpoint, sending data through UDP/IP results from system calls equivalent to *write* or *sendto*. *sendto* requires the destination address on each call; *sockargs* and *getsock* produce the socket control block associated with this operation. With this information, *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 buffers. These buffers are given to *udp_usrreq* which calls *udp_output* after a pseudo connection is established via *in_pcbconnect* with its associated routines, *in_pcblookup, in_netof, and if_ifonnetof*. As with TCP, *udp_output* represents the output processing of the UDP protocol. At this level the header is created and both the header and data are checksummed by *udp_cksum*. The header and the data are then passed to *ip_output* as in TCP/IP. *ip_output* checksums the header in *in_cksum* and passes the message to the appropriate network interface, in this case *en_output*. Again, the mbufs must be copied and mapped into a single contiguous memory space before transmission; this is done in *if_wubaput*.

From the row entries in Table 2, we can see that of the 28 different routines listed for UDP/IP, only 7 present processing costs which vary significantly with the amount of data sent. The processing time of the other 21 routines remains practically constant. In contrast, *m_freem* exhibits its largest processing time for datagrams of size 1024 bytes. The three calls which show a larger variation in the vicinity of the 1024 bytes region are *uiomove*, as before with TCP/IP, *sosend*, and *if_wubaput*. The first two are associated with the buffer management strategy, and the latter with passing the data to be transmitted to the hardware interface in one contiguous piece. The four calls which have a larger impact in the processing of datagrams in UDP/IP are *udp_cksum*, *sosend*, *if_wubaput*, and *Copyin*. For UDP/IP, checksumming is, across most datagrams sizes, the single most expensive operation performed. As mentioned in Section 4.1, redundancy of this operation should be avoided if the Ethernet is reliable [20-21]. (However, TCP cannot avoid it.) For larger datagram sizes, the greatest impact comes from those routines which do copying of data across the interfaces. *sosend*, as was also the case in TCP/IP, is an important factor in the time spent processing messages.

Table 2 also shows that for UDP/IP's implementation the processing costs are somewhat more evenly distributed across many routines than for TCP/IP. This suggests that if checksumming is eliminated, speeding up more UDP/IP will require streamlining many routines. In UDP/IP there appear not to be many significant gains to be obtained from any one optimization.

4.3. Dynamic Profile of both Protocols

We can view 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. We first note that *sosend* is called by each protocol exactly 10,000 times. Moreover, *uiomove*, *ip_output*, *enoutput*, and *if_wubaput* are called about the same number of times for user data amounts of 1 through 1024 bytes. Indeed no buffering occurs for TCP since *ip_output* is called about the same number of times.

The 1025 byte size behaved quite differently. The most surprising behavior was *uiomove* in TCP/IP, where the count, instead of dropping to something in the order of 20,000, remained very high. This is so because of the 'stream' communication nature of TCP. As there are no record boundaries, the use of 'odd' send sizes causes *sosend* to copy user data in a number of small segments once the send queue reached the last 1024 bytes of buffer space allocated. If there is more than 1024 bytes of data to be transmitted but less than 1024 bytes of buffer space remaining for the socket, *sosend* will send the remaining amount using mbufs. This is exactly what happened in the 1025 byte case as witnessed by *uiomove* which is called once for each mbuf used. It is interesting to note that the UDP/IP implementation is more immune than the TCP/IP implementation to packet size changes, with respect to the number of times individual routines are called. This is mostly due to the fact that UDP preserves record boundaries and has no flow control provisions. In TCP, estimating the amount of data to copy into mbufs based solely on the amount of buffer space currently available is the problem, in similitude with the 'silly window syndrome'. In the UDP/IP implementation, however, *m_freem* and *m_get* exhibit a peak of activity for the 1024

TCP/IP Routines and System Calls	Message Size in Bytes					
	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**
ipintr	**356**	**1,532**	**1,686**	**10,035**	**10,238**	**10,012**
tcp_usrreq	10,011	10,020	10,021	10,119	10,022	10,046
tcp_input	**323**	**1,472**	**1,645**	**10,095**	**10,243**	**10,012**
tcp_output	10,010	10,017	10,015	10,104	10,008	10,015
tcp_xoutput	**323**	**1,472**	**1,645**	**10,095**	**10,243**	**10,012**
sbappend	10,000	10,000	10,000	10,088	10,000	10,000
ip_output	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,081**
tcp_cksum	**10,333**	**11,497**	**11,671**	**20,234**	**20,256**	**30,037**
in_cksum	**10,384**	**11,628**	**11,730**	**20,494**	**20,356**	**30,369**
m_copy	**10,007**	**10,022**	**10,024**	**10,136**	**10,010**	**20,022**
enoutput	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,081**
enstart	**10,024**	**10,059**	**10,037**	**10,188**	**10,057**	**20,141**
if_wubaput	**10,016**	**10,049**	**10,032**	**10,182**	**10,024**	**20,056**
in_lnaof	**30,075**	**30,190**	**30,131**	**30,644**	**30,136**	**60,361**

Table 3: Number of Calls per Function for 10,000 TCP/IP Transmissions.
[*Highlighted in boldface are those calls with large variations.*]
[*Highlighted in italics are those calls with identical counts.*]

byte case which contrasts with all other sizes. This is due to the handling of trailer protocol packets.

5. Conclusions

For users who want to implement distributed applications based on Berkeley UNIX 4.2BSD computing environments interconnected through Ethernets, the system currently provides two basic transmission protocols for interprocess communication: TCP and UDP. Both, in turn, are based on IP for actual data transmissions. This paper has presented a microanalysis of the dynamic behavior that the current implementation of these protocols exhibits. Even though there are currently other protocol families at different stages of implementation which will coexist with the above protocols in the kernel of Berkeley UNIX, this paper has only addressed performance issues relating to TCP/IP and UDP/IP.

Let us define user process network latency to be the minimum time required to send a single byte of data. When the ether and both the sending and receiving hosts had no other user activities in them but our tests, complementary results [2] show that, for the VAX 11/750, the latency for TCP/IP is approximately 5.5 milliseconds, and for UDP/IP is 6.4 milliseconds. Sending 1024 bytes took 13.3 milliseconds with TCP/IP and 7.8 milliseconds with UDP/IP. We thus see that the transmission cost per byte is substantially lower for 1024 byte messages. Moreover, because of packet acknowledgements TCP/IP is sensitive to round trip network time while UDP/IP is not.

A detailed protocol implementation analysis has been presented for TCP/IP and UDP/IP. For TCP/IP, those routines which do the copying of data appear to make preponderant contributions to the total elapsed time (see Table 1). For UDP/IP, the single most expensive operation is

UDP/IP Routines and System Calls	Message Size in Bytes					
	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*
sockargs	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
getsock	*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*
m_freem	**19,003**	**19,406**	**19,204**	**19,331**	**29,194**	**19,848**
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*
in_pcbconnect	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_pcblookup	10,052	10,055	10,049	10,089	10,069	10,073
in_netof	40,057	40,051	40,075	40,108	40,075	40,111
if_ifonnetof	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_pcbdisconnect	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
udp_output	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
m_get	**20,023**	**20,021**	**20,030**	**20,040**	**30,029**	**20,041**
ip_output	10,019	10,017	10,025	10,036	10,025	10,057
udp_cksum	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*	*10,000*
in_cksum	29,021	28,820	28,458	28,710	29,818	29,770
in_lnaof	30,108	30,105	30,122	30,196	30,143	30,183
ipintr	8,996	9,403	9,211	9,347	9,888	9,850
enrint	8,255	8,503	8,298	8,369	8,117	8,136
enoutput	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
if_wubaput	10,019	10,017	10,025	10,036	10,025	10,037
getf	*10,840*	*10,840*	*10,840*	*10,840*	*10,840*	*10,840*

Table 4: Number of Calls per Function for 10,000 UDP/IP Datagrams.
[*Highlighted in boldface are those calls with large variations.*]
[*Highlighted in italics are those calls with identical counts.*]

the computation of the checksums (see Table 2). For both protocols, the buffer scheme used in the implementation appears to have an overwhelming effect on performance. Since UDP/IP has no flow control, it sends data atomically and only limits a packet's maximum size. This protocol is not so sensitive to varying data sizes. The drastic increases in overhead in the routines shown in bold appear to be due to the data buffer management scheme chosen. On the other hand, TCP/IP, with its windows and data streaming, is sensitive to the amount of data presented. Thus, in addition to the increased overhead seen in the routines which deal with data within the buffer management system, the actual protocol implementation overhead appears to increase noticeably because of the varying amounts of data presented.

The miscoordination of buffering between different layers with independent data placement policies can lead to severe inefficiencies. This is best exemplified by the behavior of *sosend* when transmitting 1025 byte messages under TCP/IP. The strategy of completely filling the send buffer, despite the induced buffer fragmentation, rather than waiting for additional buffer availability is inadequate. This is completely analogous to the 'silly window syndrome'.

6. Epilogue

Since this study was conducted several changes have been made to the implementations of TCP/IP and UDP/IP, as well as to the buffer management policies and default buffer sizes. These changes will be present in future BSD releases. We highlight some.

The buffer size at the socket level is now a settable parameter. When increased to 8 kilobytes we observed an improved throughput for TCP/IP in the order of 20%. In TCP, a facility has been added for coalescing messages to be sent while waiting for acknowledgement of outstanding packets. This minimizes, in the case of receiver busy, the number of transmissions between it and any sender. *sosend* has been changed so as to align 1024 byte messages whenever possible, delaying if necessary. From the receiver end of transmissions, and having in mind that the processing of acknowledgements consumes a substantial amount of processor resources, the scheme for delayed acknowledgements has been tuned. This scheme works best with a larger socket buffer size. For UDP, routing has been enhanced to cache the last computed route; if two consecutive datagrams go to the same destination, the route for the second need not be computed. This routing change should improve throughput for multiple datagrams to the same destination. A complete assessment of these changes has yet to be made.

7. Bibliography

[1] Almes, G. T., and Lazowska, E. D., "The Behavior of Ethernet-like Computer Communications Networks", Proceedings of the 7th Symposium on Operating System Principles, 1979, pp. 66-81.

[2] Cabrera, L. F., Hunter, E., Karels, M. J., and Mosher, D., "A User-Process Oriented Performance Study of Ethernet Networking Under Berkeley UNIX 4.2BSD," Report No. UCB/CSD 84/217, University of California, Berkeley, December 1984, pp. 1-37.

[3] Cheriton, D., and Zwaenepoel, W., "The Distributed V Kernel and its Performance for Diskless Workstations", Proceedings of the 9th SOSP, November 1983.

[4] Gonsalves, T. M., "Packet-Voice Communication on an Ethernet Local Computer Network: an Experimental Study", Proceedings of the SIGCOMM 1983 Symposium on Communication Architectures and Protocols, Austin, Texas, March 1983, pp. 178-185.

[5] Graham, S. L., et. al., "An Execution Profiler for Modular Programs", Software – Practice and Experience, Vol. 13, 1983, pp. 671-685.

[6] Hagmann, R. B., "Performance Analysis of Several Backend Database Architectures", Ph.D. Thesis, Report No. UCB/CSD 83/124, University of California, Berkeley, August 1983.

[7] Hunter, E., "A Performance Study of the Ethernet Under Berkeley UNIX 4.2BSD", Proceedings of CMG XV, San Francisco, California, December 1984, pp. 373-382.

[8] Lazowska, E. D., et. al., "File Access Performance of Diskless Workstations", Technical Report 84-06-01, June 1984, Department of Computer Science, University of Washington.

[9] Leffler, S. J., and Fabry, R., "A 4.2BSD Interprocess Communication Primer", Report No. UCB/CSD 83/145, University of California, Berkeley, July 1983.

[10] Leffler, S. J., et. al., "4.2BSD Networking Implementation Notes," Report No. UCB/CSD 83/146, University of California, Berkeley, July 1983.

[11] Leffler, S. J., et al, "Measuring and Improving the Performance of 4.2BSD", Proceedings of the Summer 1984 USENIX Conference, Salt Lake City, June 1984, pp. 237-252.

[12] Leffler, S. J., and Karels, M. J., "Trailer Encapsulations", RFC 893, Network Working Group, University of California, Berkeley, April 1984.

[13] Metcalfe, R. M., and Boggs, D. R., "Ethernet: Distributed Packet Switching for Local Computer Networks", CACM, Volume 19, Number 7, July 1976, pp. 395-404.

[14] Padlipsky, M., "TCP-ON-A-LAN", RFC 872, USC Information Sciences Institute, September 1982.

[15] Postel, J., "User Datagram Protocol", RFC 768, USC Information Sciences Institute, August 1980.

[16] Postel, J., "Internet Protocol - DARPA Internet Program Protocol Specification", RFC 791, USC Information Sciences Institute, September 1981.

[17] Postel, J., "Transmission Control Protocol", RFC 793, USC Information Sciences Institute, September 1981.

[18] Rajaraman, M. K., "Performance Measures for a Local Network", ACM Sigmetrics Performance Evaluation Review, Volume 12, Number 2, Spring-Summer 1984, pp. 34-37.

[19] Sechrest, S., "Tutorial Examples of Interprocess Communication in Berkeley UNIX 4.2BSD", Report No. UCB/CSD 84/191, University of California, Berkeley, June 1984.

[20] Shoch, J. F., and Hupp, J. A., "Performance of an Ethernet Local Network -- A Preliminary Report", Proceedings of Spring COMPCON 80, San Francisco, February 1980.

[21] Shoch, J. F., and Hupp, J. A., "Measured Performance of an Ethernet Local Network", CACM, Volume 23, Number 12, December 1980, pp. 711-721.

[22] Terry, D., and Andler, S., "Experience With Measuring Performance of Local Network Communications", IBM San Jose Research Laboratory Research Report RJ 3743 (43119), December 1983, pp. 1-6.

[23] "UNIX Programmer's Manual", 4.2 Berkeley Software Distribution, Virtual VAX-11 Version, Computer Science Division, University of California at Berkeley, August 1983.

8. Appendix A

8.1. Software for TCP/IP Assessment (Sender)

```
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in address;
char sendbuf[1025];;

main(argc,argv)
char *argv[];
{
        int i;
        struct hostent *phostent;
        int des;
        int size;

        des = socket(AF_INET,SOCK_STREAM,0);
        if (des < 0 )
                perror("socket"),exit(-1);
        phostent = gethostbyname(argv[1]);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr  = *(int *)phostent->h_addr;
        address.sin_port = 4321;
        if (connect(des,&address,sizeof(address)))
                perror("connect"), exit(-1);
        size = atoi(argv[2]);
        for (i=0; i<10000; i++)
                write(des,sendbuf,size);
}
```

8.2. Software for UDP/IP Assessment

```
#include <sys/types.h>
#include <netdb.h>
#include <sys/socket.h>
#include <netinet/in.h>

struct sockaddr_in address;

main(argc,argv)
char *argv[];
{
        int s, i;
        char buf[1025];
        struct hostent *phostent;
        int size;
        extern errno;

        if((s = socket(AF_INET,SOCK_DGRAM,0))  == -1) perror("socket") ;
        phostent = gethostbyname(argv[1]);
        address.sin_family = AF_INET;
        address.sin_addr.s_addr  = *(int *)phostent->h_addr;
        address.sin_port = ntohs(1234);
        size = atoi(argv[2]);
        printf("Testing byte size of %d0,size);
        for(i = 0; i < 10000; i++)
                sendto(s,buf,size,0,&address,sizeof(address));
        if (errno) perror("udp: ");
}
```