Is Layering Harmful?

Remote Procedure Call mechanisms over TCP can produce behavior analagous to the Silly Window Syndrome because of a mismatched interface between the socket and the TCP modules.

Jon Crowcroft, Ian Wakeman, Zheng Wang, and Dejan Sirovica

When invoking an operation on a remote machine, the paradigm with which programmers are most comfortable is the Remote Procedure Call (RPC). In this operation, the local process invokes a stub procedure, which marshalls the parameters to be passed to the remote operation into a machine-independent format, then sends the parameters and a request identifying the operation to be invoked to the remote machine. The operation is performed, and the results are returned to the local stub procedure, which passes the results to the invoking program. If the link to the remote machine is not totally reliable, then in order to create the once-only semantics that most calls require, mechanisms that guarantee the correct delivery of the data must be invoked. Where these mechanisms are placed is a matter of debate; the Sun Micro Systems Network File System™ is best known for using its own reliability mechanisms over the User Datagram Protocol (UDP) [1]. However, building reliable delivery mechanisms that can cope with arbitrary message sizes as well as the problems of duplicates, missing messages and out-of-order delivery is not a trivial task, and if it is constructed wrongly, performance suffers—not only that of the operation invoking the messages, but that of the other denizens of the network as well.

An alternative approach is to use the reliable delivery mechanisms that already exist in most systems, such as the Open Systems Interconnect (OSI) Class 4 Transport Protocol (TP4) [2], or the Department of Defense (DoD) Transmission Control Protocol (TCP) [1]. These protocols ensure not only reliable delivery of data, but also responsible use of the network resource.

There are concerns that RPC is not the correct mechanism to use in a distributed system that runs over high bandwidth-delay-product networks, as the block-on-response semantics of the call are an inefficient use of resources. However, it is an easy mechanism for programmers raised on single-processor systems to use in building distributed systems, and it will be sufficiently pervasive to make optimizing its performance worthwhile.

As part of an investigation into the performance of the Sun Micro Systems RPC mechanism [3] over TCP, we found unexpected glitches in the performance of RPC calls as their size increased. This aroused curiosity about where these glitches were arising from. Eventually, investigations led to a mismatched interface between layers of a protocol stack, and an inappropriate buffering strategy between the socket code and the TCP code. A diagram of the protocol stack can be seen in Table 1.

Although correcting these deficiencies is an important result in its own right, we feel that the problems we have uncovered illustrate the deficiencies in the models used to implement communications systems. The concentration on a layered architecture in which the functions of each layer are independent of each other results in a common processing path for incoming and outgoing data, which passes along a path with crevasses and cliffs between layers. Instead, we recommend the use of a separate model for designing and implementing a real system, in which the focus of design is on supporting the requirements of the applications' data unit.

The first part of this article shows the detection and diagnosis of the problem, and the second part provides some pointers to a design approach that could avoid the problems of mismatched communications layers.

This Graph Looks Strange . . .

The RPC program used to construct the test was a very simple echo program, where the data passed was turned around and echoed back. We ran the program for a fixed number of procedure calls and recorded the time taken while varying the size of the data to be echoed. The machines were Sun SLCs running SunOs 4.1, connected via an Ethernet. Since the TCP connection is maintained while the calls are being made, the latency of connection setup and teardown is amortised over all the calls. A graph showing the performance we discovered can be seen in Fig. 1. Similar behavior was observed on Sun3 and HP400 machines. The variations in thresholds and per-

Jon Crowcroft is a senior lecturer in the Department of Computer Science, University College London.

Ian Wakeman is with UCL as a researcher.

Zheng Wang is working toward his Ph.D at University College London.

Dr. Sirovica is a member of technical staff at USWEST Advanced Technologies.

™Sun Micro Systems and Network File System (NFS) are trademarks of Sun Micro Systems.

0163-6804/92/$03.00 1992© IEEE

formance between the various machines were minor.

We were very puzzled by the order-of-magnitude glitch that started at around 4,000 bytes of data and finished 1,000 bytes later, and by the irregular graph of the subsequent behavior.

Our initial thoughts were that the problem was an aberrant interaction between the windowing flow control of TCP and the buffer sizes of the RPC call. To investigate this, we ran tcpdump on a third machine on the same ethernet as the machines carrying the aberrant conversation. Tcpdump is a traffic-monitoring program written by Van Jacobson *et al.* [4], which can capture traffic and print out the constituent packets with the following information (shown pictorially in Table 2):

- Time: This is the time in which the packet traversed the Ethernet, accurate to about +/- 10 ms using the Network Interface Tap in Sunos 4.1.1.
- src, dst: These are the source and destination Internet Protocol (IP) address and TCP/UDP port number, and can be used to deduce application in most cases using the well known port concept in the IP Architecture.
- flgs: These flags indicate whether the packet is the start or end of a connection, or whether the packet has a PUSH bit set if TCP data.
- seq: If TCP, the start and ending (byte) sequence number of this packet.
- lth: The packet length in bytes.
- ack: If TCP, the sequence number that this packet acknowledges.
- win: The size of the receive window in bytes that the sender of this packet is advertising.

The sample trace displayed in Table 3 shows the pattern of packet transmission for data sizes of 4,800 bytes. The initial UDP packets query the portmapper as to which port to use. The trace is displayed in a time sequence diagram in Fig. 2. The other tool we used to investigate the problem was the trace facility, which intercepts the system calls and signals of a program and displays their arguments and results [5]. In this way, we discovered that the rpcgen[1]-generated code uses user space buffers of 4,000 bytes for External Data Representation (XDR) conversion [6]. These buffers are then submitted to the kernel for copying to mbufs and then onto the socket queue. A curtailed version of the output from the trace command is shown in Table 4. The write and read calls are shown in bold.

As can be seen in Table 3, the default window size of the TCP connection is 4,096 bytes. If the buffer used to pass data were 4,096 bytes as well, then each buffer would be sent as a full window, and we would have seen a smoother graph for the transfer of data. However, they were not, and we saw suboptimal behavior when the sizes of the buffers were not matched. Thus, the first lesson is to match buffer sizes whenever possible, so that there are never any small amounts of space left over. (Of course, we could not possibly speculate that the reason for having 4,000-byte buffers in the XDR code was because someone interpreted "4K" as meaning "4,000" and not "4,096.") However, the behavior still required a deeper explanation, so we started looking at the tcpdump traces more closely.

The length of the RPC call was extended beyond that expected by the delays between the small packet in Fig. 2, carrying sequence numbers 4001:4097, its corresponding acknowledgement, and then the larger data packet carrying sequence numbers 4,097:4,837. These packets are highlighted in bold in Fig. 2. The gaps in between sending the small packet and the succeeding packet was long enough to force the data to be read in two separate reads, one of 96 bytes and the other of 756 bytes.

Code	OSI layer	Operating System Layer
Client caller code	Application	User Space
Client Stub		
XDR	Presentation	
Socket code	Session	Kernel
TCP code	Transport	
IP	Network	
Net	Data Link	

■ **Table 1.** *Protocol Stack of the RPC call*

time	src	dst	flgs	seq	lth	ack	win

■ **Table 2.** *Format of tcpdump trace*

```
0.140000 camus.1096>sartre.sunrpc:udp 56
0.150000 sartre.sunrpc>camus.1096:udp 28
0.150001 camus.1043>sartre.1055:S 1:1(0) win 4096<mss 1460>
0.150002 sartre.1055>camus.1043:S 1:1(0) ack 1 win 4096 <mss 1460>
0.150003 camus.1043>sartre.1055:.ack 1 win 4096
0.160000 camus.1043>sartre.1055:.1:1461(1460) ack 1 win 4096
0.160001 camus.1043>sartre.1055:.1461:2921(1460) ack 1 win 4096
0.160002 camus.1043>sartre.1055: P 2921:4001(1080) ack 1 win 4096
0.160003 sartre.1055>camus.1043:.ack 4001 win 4096
0.160004 camus.1043>sartre.1055:P4001:4097(96)ack 1 win 4096
0.170000 sartre.1055>camus.1043:.ack4097 win 4000
0.180000 camus 1043>sartre.1055:P 4097:4853(756)ack 1 win 4096
0.230000 sartre.1055>camus.1043:.1:1461(1460)ack 4853 win 4096
0.230001sartre.1055>camus.1043:.1461:2921(1460)ack4853 win 4096
0.230002 sartre.1055>camus.1043:P2921:4001(1080) ack 4853 win 4096
0.230003 camus.1043>sartre.1055:.ack 4001 win 1568
0.230004 camus.1043>sartre.1055:.ack4001 win 4096
0.230005 sartre.1055>camus.1043:P4001:4097(96) ack 4853 win 4096
0.430000 camus.1043>sartre.1055:.ack4097 win 4096
0.430001 sartre.1055>camus.1043:P 4097:4837(740) ack 4853 win 4096
0.490000 camus.1043>sartre.1055:F 4853:4843(0) ack 4837 win 4096
0.490001 sartre.1055>camus.1043:.ack 4854 win 4096
0.490002 sartre.1055>camus.1043:F 4837:4837(0) ack 4854 win 4096
0.490003 camus.1043>sartre.1055:.ack 4838 win 4096
```

■ **Table 3.** *tcpdump traces for buffer size of 4800 bytes (with 52 bytes of rpc overhead)*

```
socket (2,1,6) = 4
bind (4,""..,16) = -1 EACCES (Permission denied)
connect (4,"".., 16) = 0
gettimeofday (0xf7fff8c0,0) = 0
getpid () = 865
brk (0xd4c0) = 0
gettimeofday (0xf7fff9c0,0) = 0
write (4""..., 4000) = 4000
write (4"200 3Paaaaaaaaaaaaaaaaaaaaaaaaaaaa"..,852) = 852
select (64, 0xf7fff5c8, 0, 0, 0xac40) = 1
read (4, ""..., 4000) = 4000
brk (0xf4c)) = 0
select (64, 9xf7fff5c8, 0, 0, 0xac40) = 1
read (4,"200 3@aaaaaaaaaaaaaaaaaaaaaaaaaaa"..,4000) = 96
select (64, 0xf7fff650,0,0, 0xac40) = 1
read (4, "aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa"..., 4000) = 740
gettimeofday (oxf7fff9c0,0) = 0
close (0) = 0
close (1) = 0
close (2) = 0
exit (0) = ?
```

■ **Table 4.** *Sample trace output for the RPC call with 4800 bytes*

[1] rpcgen is a tool that generates the stub routines for the rpc code, and incorporates the functions necessary for Sun XDR processing (presentation conversion) and communication programming.

```
0.920001 camus.1047>sartre.discard: S 1:1(0) win 4096
0.920002 sartre.discard>camus.1047:S 1:1(0) ack 1 win 4096 <mss
1460>
0.920003 camus.1047> sartre.discard:.ack 1 win 4096
0.950000 camus.1047>sartre.discard:.1:1461(1460) ack 1 win 4096
0.950001 camus.1047> sartre.discard:.1461:2921(1460) ack 1 win 4096
0.950002 camus.1047> sartre.discard: P 2921:4096(1175) ack 1 win
4096
0.960000 sartre.discard>camus.1047:.ack 4096 win 2049
0.960001 camus.1047<sartre.discard: P 4096:4097(1) ack 1 win 4096
0.960002 sartre.discard>camus.1047:.ack4096 win 4096
1.010000 camus.1047>sartre.discard: FP 4097:5119(1022) ack 1 win
4096
1.010001 sartre.discard>camus.1047:.ack 5120 win 3074
1.020001 sartre.discard>camus.1047:F1:1(0) ack 5120 win 4096
1.020002 camus.1047>sartre.discard:.ack 2 win 4096

0.470001 camus.1048>sartre.discard:S 1:1(0) win 4096 <mss 1460>
0.470002 sartre.discard>camus.1048:S 1:1(0) ack 1 win 4096 <mss
1460>
0.470003 camus.1048>sartre.discard:.ack 1 win 4096
0.500000 camus.1048>sartre.discard:.1:1461(1460) ack 1 win 4096
0.500001 camus.1048>sartre.discard:.1461:2921(1460) ack 1 win 4096
0.500002 camus.1048>sartre.discard:P 2921:4096(1175) ack 1 win 4096
0.500003 sartre.discard>camus.1048:.ack 4096 win 4096
0.510000 sartre.discard>camus.1048:.ack 4096 win 4096
0.510001 camus.1048>sartre.discard:P 4096:5121 (1025) ack 1 win 4096
0.550000 camus.1048>sartre.discard:F 5121:5121 (0) ack 1 win 4096
0.550001 sartre.discard>camus.1048:.ack 5122 win 4096
0.570000 sartre.discard>camus.1048: F 1:1(0) ack 5122 win 4096
0.570001 camus.1048>sartre.discard:.ack 2 win 4096
```

■ **Table 5.** *tcpdump traces for a buffer size of 4095 bytes, and data sizes of 5118 bytes and 5120 bytes*

```
0.144713 spasky.1160>fischer.discard: S 1:1(0) win 4096 <mss 1460>
0.145877 fischer.discard>spasky.1160: S 1:1(0) ack 1 win 4096 <mss
1460>
0.146467 spasky.1160>fischer.discard:.ack 1 win 4096
0.155208 spasky.1160>fischer.discard:.1:1461(1460) ack 1 win 4096
0.156432 spasky.1160>fischer.discard:.1461:2921(1460) ack 1 win 4096
0.157292 spasky.1160>fischer.discard:P2921:4096(1175)ack 1 win 4096
0.157642 spasky.1160>fischer.discard:P4096:4097(1) ack 1 win 4096
0.180476 fischer.discard>spasky.1160:.ack 4097 win 2048
0.181374 fischer.discard>spasky.1160:.ack 4097 win 4096
0.182697 spasky.1160>fischer.discard:P 4097:5119(1022) ack 1 win 4096
0.195328 spasky.1160>fischer.discard: F 5119:5119(0) ack 1 win 4096
0.195798 fischer.discard>spasky.1160:.ack 5120 win 4906
0.204607 fischer.discard>spasky.1160: F 1:1(0) ack 5120 win 4096
0.205308 spasky.1160 > fischer.discard:.ack 2 win 4096

0.381148 spasky.1161>fischer.discard:S 1:1(0) win 4096 <mss 1460>
0.382277 fischer.discard>spasky.1161:S 1:1(0) ack 1 win 4096 <miss
1460>
0.382906 spasky.1161>fischer.discard:.ack 1 win 4096
0.391533 spasky.1161>fischer.discard:.1:1461(1460)ack 1 win 4096
0.392763 spasky.1161>fischer.discard:.1461:2921(1460) ack 1 win 4096
0.393605 spasky.1161>fischer.discard:P 2921:4096(1175) ack 1 win 4096
0.416785 fischer.discard>spasky.1161:.ack 4096 win 2049
0.417739 fischer.discard>spasky.1161:.ack 4096 win 4096
0.418893 spasky.1161>fischer.discard:P 4096:5121(1025) ack 1 win 4096
0.431728 spasky.1161>fischer.discard:F 5121:5121(0) ack 1 win 4096
0.432271 fischer.discard>spasky.1161:.ack 5122 win 4096
0.441032 fischer.discard>spasky.1161:F1:1(0) ack 5122 win 4096
0.441761 spasky.1161>fischer.discard:.ack 2 win 4096
```

■ **Table 6.** *tcpdump traces for a buffer size of 4095 bytes, and data sizes of 5118 bytes and 5120 bytes with NO_TCPDELAY*

The cost of the additional system call and its concomitant context switches accounted for the additional time spent servicing the RPC call. It thus appeared that the performance glitch was caused by problems in the send and receive paths of the operating system, and not necessarily by the rpcgen code. In particular, the problems seemed to be caused by the splitting of the second data buffer written to the send code into a small and larger packet, even though the total data to be sent is less than the maximum size of an Ethernet packet.

To ensure that the problem was independent of the rpcgen code, we hacked a program that wrote a variable size of buffer and a variable amount of data to the discard port, and used this to discover what was happening at the TCP level. Investigation of the performance showed that the falling edge on the glitch lay between 5,118 and 5,120 bytes of data being sent.

As shown in Table 5 and pictorially in Figs. 3 and 4, it is only the smaller of two data sizes that generates a small packet, which causes a slowdown of the data flow due to use of the Nagle algorithm [7]. This algorithm attempts to reduce the congestion in the network by only allowing one unacknowledged small packet in the network at any one time. Unfortunately, this algorithm interacts quite badly with the delayed acknowledgment processing that occurs at the other end of the connection. After the small packet is sent, we have insufficient data remaining to make a "large" packet, so this packet cannot be sent until the previous small packet is acknowledged, thus obeying Nagle's edict. However, the receive side of TCP processing delays generation of acknowledgements when data is received, since there is likely to be more data following soon. So, when a small packet is sent (and no other data), there will be a long wait before the acknowledgment is generated and the throughput will suffer. When we repeated the experiment with the Nagle delay turned off at the sender (TCP_NODELAY as a socket option), we gained the results in Table 6.

As can be seen, although the delay between packets has been reduced, the data is still transferred for the smaller data size in a small and a big packet. In addition, the delayed acknowledgment policy is not configurable, so the overall time spent transferring the data is still dominated by the timeout for the generation of the acknowledgment and only slightly improved by the faster generation of the larger packet. This is confirmed by repeating the initial RPC experiment, which showed little improvement with using the TCP_NODELAY option.

However, this effect is only apparent for transferring small amounts of data, as the graph in Fig. 5 shows. This graph tracks sequence number versus time for buffer sizes of 4,095 bytes and 4,096 bytes, over a transfer of one million bytes (Nagle is on). The gradient shows the throughput achieved in the transfer, which is identical in both cases.[2] Obviously, the problem we are searching for is a boundary effect, with no impact on large transfers but important for RPC calls.

Tracking the Small Packet

Looking through the TCP code to trace where a small packet could have been sent, we noticed that the PUSH bit is only set on a packet when it is the last data left in the send queue. Now, since the small packet has the PUSH bit set, we logically deduced that the socket code had placed a

2 This is to be expcected as the TCP code has been heavily optimized for bulk throughput transfers.

small amount of data in the send queue for the case when it had less than a certain threshold amount of data to send, and had not placed any data in the queue when it had more than the threshold amount of data. This explains why bulk data transfers are not affected by this phenomenon, since each write from user space will have a large amount of data for the sosend code to place in the queue, and the amount of user data should not drop below the threshold.

To confirm this, we went to the BSD socket code to see when data is placed on the send queue. First we looked at the Tahoe release of the 4.3 BSD code, in which we found the code in Table 7.

This appeared to be the root of our problems, in that a small fraction of data can be written when the total data is less than the size of an mbuf cluster. We found this to be analagous to the Silly Window Syndrome for TCP described in [8], where throughput of a connection is limited by only having a small window open in the flow control mechanism, so that less-than-optimal-size packets of data are sent throught the network. The problem is solved for TCP by preventing receivers from advertising small increases in receive windows, and by transmitters refraining from sending if the advertised window is too small. In the problem presented here, the flow control is between the socket layer and the TCP layer, and it is the sender—the socket layer—who is misbehaving and sending small chunks of data when it is more efficient to send larger chunks. The socket layer should refrain from placing more data into the buffer until there is sufficient space[3] to send a larger chunk of data.

It is important to realize the significance of where this error originated and, in particular, how the layered structure of the software contributed to the problem. The communications software is implemented in a layered fashion, where each layer contains functionality independent of that in the other layers. Thus, the application processes are independent of the presentation processing and the semantics of the data passed to the XDR routines are hidden from the XDR processing layer. Each layer "does its thing" on the data to be transferred, and then moves it down to the layer below. In this way, we get the time sequence diagram show in Fig. 6, where the data passes from application to RPC stub routine to XDR routine (presentation layer in the International Standards Organization—ISO—Reference Model) to socket (session layer) to TCP (transport layer) and so on.

The most recent release of the BSD socket code (Reno) corrects the problem by the use of a low-water mark, in which data is only appended to the send queue when the space remaining is above a settable low-water mark (through the setting of socket options). It appears that the current default is 0 bytes, but it is suggested that this is increased to at least the default maximum segment size for TCP sockets. Users cannot be relied upon to set the proper sizes for common working of their code. A somewhat terse explanation of the thinking behind the BSD communications code can be found in [9].

To test out our hypothesis, we rebuilt a UNIX kernel with a modified sosend function that only placed partial user data into the send queue if there was more space in the send queue than the size of an mbuf cluster. The code is shown in Table 8. The result of running the same experiment as previously is shown in Fig. 7. The resultant curve is gratifyingly smooth compared to the previous result, shown by the dotted line.

```
/*
*Copyright (c) 1982, 1986 Regents of the University of California
*All rights reserved.
*
*Redistribution and use in source and binary forms are permitted
*provided that the above copyright notice and this paragraph are
*duplicated in all such forms and that any documentation,
*advertising materials, and other materials related to such
*distribution and use acknowledge that the software was developed
*by the University of California, Berkeley. The name of the
*university may not be used to endorse or promote products derived
*from this software without specific prior written permission.
*THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY
*EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT
*LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY
*AND FITNESS FOR A PARTICULAR PURPOSE.
*
*@(#uipc_socket.c7.10(Berkeley) 6/29/88
*
*Code to decide whether to write data to the send queue or not
*Nb This code has been edited to show the required features of it-
*no guarantees are given regarding its completeness
*/
/*find out how much space there is in the buffer */
        space = sbspace(&so->so_snd);

        /*
        *Block if we want to send all at once, and there is not
        *enough room
        */
        if ( (sosendallatonce(so)&&
        space<uio->uio_resid + rlen)ÔÔ
        /* Or if we have more than MCLBYTES of data and less than
        *MCLBYTES of space and there's already data in the queue
        */
        (uio->uio_resid>=MCLBYTES && space <MCLBYTES &&
        so->so_snd.sb_cc>=MCLBYTES &&
        (so->so_state & SS_NBIO)==0)){

        sbunlock(&so->so_snd);
        sbwait(&so->so_snd);
        splx(s);
        goto restart;
        }
```

■ **Table 7.** *Old sosend code that decides whether to wait on a socket buffer*

What Did We Learn?

Our problem finally resolved into a data transfer problem between two layers of a communications stack implementation. It is our contention that this problem is inherent in the design methods that are used for implementing communications software, where the conceptual model of a layered stack is also used for the engineering of the implementation. Layering is about modularizing the functions performed on data during its transfer from one machine to another, so that the complexity of the transfer can be managed. However, the flip side to modularization and data-hiding is that tuning the efficiency of the data path for transfer of data becomes difficult, as important details such as buffer sizes are hidden from each layer. Vertical partitioning emphasises the discontinuities in the data path, which then obstruct the application from receiving the quality of service it requires.

[3]Liveness of the flow control mechanism is given by the liveness of the TCP implementation; as long as the TCP mechanism works it will eventually deliver the data currently in the buffer, and free space for larger chunks of data to be placed in the buffer.

```
/*
*Copyright (c) 1982, 1986, 1988, 1990 Regents of the University of Cali-
fornia.
*All rights reserved
*
*Redistribution is only permitted until one year after the first shipment
*of 4.4BSD by the Regents.  Otherwise, redistribution and use in source
and
*binary forms are permitted provided that: (1) source distributions
retain
*this entire copyright notice and comment, and (2) distribution includ-
ing
*binaries display the following acknowledgement: This product includes
*software developed by the University of California, Berkeley and its
*contributors" in the documentation or other materials provided with
the
*distribution and in all advertising materials mentioning features or use
*of this software.  Neither the name of the University nor the names of
*its contributors may be used to endorse or promote products derived
from
*this software without specific prior written permission.
*THIS SOFTWARE IS PROVIDED "AS IS" AND WITHOUT ANY
*EXPRESS OR IMPLIED WARRANTIES, INCLUDING, WITHOUT
*LIMITATION, THE IMPLIED WARRANTIES OF MERCHANTIBILITY
*AND FITNESS FOR A PARTICULAR PURPOSE.
*
*@(#)uipc_socket.c7.23 (Berkeley) 6/29/90
*
*Code to decide whether to write data to the send queue or not
*Nb this code has been edited to show the required features of it -
*no guarantees are given regarding its completeness
*/
*If there is less space in the send queue than we have data to send*/
        if (space <resid +clen &&
        /*
        * and either mustn't be broken up OR
        * less space than the low water mark ie avoid Silly Window
        *       Syndrome
        */
        (atomic ÔÔ space < so->so_snd.sb_lowat (({
        /*release lock on state */
        sbunlock (&so->so_snd);
        /*wait until there is more space */
        error = sbwait(&so_snd);
        splx(s);
        if (error)
        goto out:
        goto restart;
        }
```

■ **Table 8.** *New sosend code that decides whether to wait on a socket buffer*

The inadequacies of the layered model for implementation receive attention from Clark and Tennenhouse [10]. They propose that communications architectures move away from layers and focus on the requirements of the application as the basis. The application data unit is used as the basis for constructing the communications software from plug-in data transfer functions for presentation processing, encryption, checksumming, etc., and control functions such as flow control are done "out of band," rather like moving from stepwise decomposition to object-oriented design, where the required functions are put together with the application data unit to create the required implementation.

Efficient data transfer requires the common processing path, from application to wire to application, to be examined and taken as an integrated path. Clark uses the concept of "Integrated Layer Processing" to cover the integration of the data path so that the data transfer is efficient. In the case above, this would require the sizes of the buffers at each stage of the transfer to be matched to the size of the application data units, i.e., the size of the RPC parameters and their associated control information. To do this requires that communications software have control paths both up and down, so that the requirements of the application and higher functions can be met by the lower functions, and the constraints of the lower parts of the data path can shape the requirements of the application.

Conclusion

We have exposed a design flaw in the communications software of most UNIX systems, positioned at the interface between the socket code and TCP, although upcoming releases of UNIX should not have this problem. In doing this, we used the trace tool to discover the interactions of the interlayer communications, from user space to kernel, and the tcpdump tool to discover the interactions of the interpeer communications.

The advent of high-bandwidth networks, as well as an increased range of applications from video and voice to more traditional data transfer, is likely to increasingly reveal the inadequacies of the layered communications architecture as a model for constructing real systems. We have shown a design fault at a layer boundary that reduced efficiency; at higher speeds and with more demanding applications, any discontinuity in the data path will reduce the performance to below that which the user will find acceptable. We believe it is necessary to start adopting new models for designing communications models such as those in [10], which allow an application to specify its requirements and have them met by an integrated data transfer path.

References

[1] D. Comer, *Interworking with TCP/IP*, Prentice Hall International, 1988.
[2] ISO, "Information Processing Systems—Transport Service Definition," Int'l. Std. 8072, June 1986.
[3] Sun Microsystems Inc., "RPC: Remote Procedure Call Protocol," rfc1057, SRI_NIC, June 1988.
[4] V. Jacobson *et al.*, "TCPDUMP(1), BPF...," *UNIX Manual Page*, 1990.
[5] Sun Microsystems Inc., "Trace," *Manual Page*, Aug. 1989.
[6] Sun Microsystems Inc., "XDR: External Data Representation Standard," rfc1014, SRI_NIC, June 1987.
[7] J. Nagle, "Congestion Control in IP/TCP Internetworks," rfc896, SRI_NIC, Jan. 1984.
[8] D. Clark, "Window and Acknowledgment Strategy in TCP," rfc813, SRI_NIC, July 1982.
[9] V. Jacobson, "Tutorial on Efficient Protocol Implementation," *SIGCOMM '90*, ACM, Sept. 1990.
[10] D. Clark and D. Tennenhouse, "Architectural Considerations for a New Generation of Protocols," *SIGCOMM '90*, ACM, Sept. 1990.

Biographies

Jon Crowcroft is a senior lecturer in the Department of Computer Science, University College London, where he is responsible for a number of European and US funded research projects in Multi-media Communications. For the last two years he has been consulting to the Bloomsbury Computing Consortium as a Senior Systems Analyst on the installation of a multi-campus distributed system. He graduated in Physics from Trinity College, Cambridge University in 1979, and gained his MSc in Computing in 1981.

Ian Wakeman gained degrees in electrical engineering from Cambridge and Stanford in 1987 and 1988, then worked at the GEC Hirst Research Centre on high speed networks and their protocols. In 1991 he joined UCL as a researcher, and he is currently interested in the problems of transmitting video over packet switched networks and other areas of multi-media communications.

Zheng Wang received his B. Eng. degree in Electronic Engineering from Zhejiang University, China, in 1985. He is now working toward his Ph.D at University College London, UK.

Dejan Sirovica received his B.Sc. and D. Phil. degrees from the University of Sussex, England, in 1982 and 1988, respectively. From 1982 to 1990, he worked on SERC, ALVEY, and RACE research projects at the University College London respectively. Dr. Sirovica is currently a member of technical staff at USWEST Advanced Technologies, Boulder, Colorado.