15

Deadlock situations in TCP over ATM

Kjersti Moldeklev[a] and Per Gunningberg[b]

[a]Norwegian Telecom Research, P.O. Box 83, N-2007 Kjeller, Norway

[b]Swedish Institute of Computer Science, P.O. Box 1263, S-16428 Kista, Sweden*

Abstract

The implementation of protocols, such as TCP/IP, and their integration into the operating system environment is very decisive for protocol performance. Putting TCP on high-speed networks, e.g. ATM, with large maximum transmission units causes the TCP maximum segment size to be relatively large. What Nagle's algorithm consider a "small" segment is not small anymore which affects the TCP throughput. We report on TCP/IP throughput performance measurements for various sizes of send and receive socket buffers, using two Sun IPX machines running SunOS 4.1.1 connected to FORE System's ATM network. For some common combinations of socket buffer sizes we observe a dramatic performance drop to less than 3% of normal throughput. The drop is caused by a deadlock situation in the TCP connection which is resolved by the 200 ms spaced timer generated TCP acknowledgment. The factors which in combination force the TCP connection into deadlocks are a large maximum segment size, asymmetry of the socket buffer sizes, use of Nagle's algorithm, the delayed acknowledgment strategy, the sequence of actions on acknowledgment reception, and at last the socket layer optimization for efficient memory management. We explain what causes the deadlock situations, discuss some alternatives to avoid or prevent the situations and present measurement results from proposed changes in the implementation.

Keyword Codes: C.2.2; C.2.5; C.4
Keywords: Computer-Communication Networks, Network Protocols; Local Networks; Performance of Systems

1. INTRODUCTION

The TCP/IP protocols are often the first protocol suite to be put on high-speed networks such as ATM (Asynchronous Transfer Mode). This in spite of the fact that TCP/IP originally was not designed to match the characteristics of high-performance networks. Several extensions of the TCP protocol [1] [2] have been suggested to make it perform better over these networks and for connections with a high bandwidth-delay product [3]. New high bit-rate demanding applications, such as multimedia conference systems, and the characteristics of high-speed networks, have triggered a lot of research on new transport protocols [4].

* Email: kjersti.moldeklev@tf.tele.no, per@sics.se

G. Neufield et al. (eds.), *Protocols for High Speed Networks IV*

However, it is well known that the *implementation* of protocols, and their *integration* into the operating system environment may dominate all improvements of protocol mechanisms. Implementation optimizations for TCP/IP on low-speed networks (with smaller data units) may be totally wrong for high-performance networks. In this paper we show that some implementation optimizations for the "ethernet era" in fact degrade TCP/IP performance on high-speed networks with large data units.

We have measured the throughput performance of TCP over ATM between two Sun IPX machines running SunOS 4.1.1, for various sizes of send and receive socket buffers. Normally we measured around 20 Mbit/s sustained throughput, but for some combinations of socket buffer sizes we observed a dramatic drop to between 0.65 to 0.16 Mbit/s. This is less than 3% of the normal throughput. It happened for common combinations of socket buffer sizes, such as a send socket buffer of 16 kbytes and a receive socket buffer of 32 kbytes.

The ATM network transfers small data units, called cells, which are 53 bytes long with a 48 byte payload. For computer communication the recommended end-to-end ATM service is through an ATM Adaptation Layer (AAL), either AAL3/4 or AAL5. The adaptation layer aggregates cells into much larger data units which are more efficiently handled by higher layers and better match the application data units. The TCP/IP protocols use the size of these AAL data units - the ATM network MTU (Maximum Transmission Unit) - to compute the Maximum Segment Size (MSS) [5] [6]. In our measurements the AAL3/4 payload is 9244 bytes. The normal TCP/IP header is 40 bytes which means that the MSS used by TCP is 9204 bytes.

The dramatic drop in performance is caused by a deadlock situation in the TCP connection which is broken up by the 200 ms timer generated TCP acknowledgment. It causes TCP to behave as a stop-and-go protocol with one or two data segments sent every 200 ms. The deadlock occurs when the amount of data sent is not enough to trigger a TCP window update packet at the receiver, and at the same time there is not enough space in the send buffer to create a segment of size MSS bytes. Nagle's algorithm prohibits the sending of non-MSS segments if there are unacknowledged bytes. Since TCP piggybacks acknowledgments onto window updates, the connection is deadlocked until the receiver sends a timer generated acknowledgment.

The deadlock problem also exists for small MSS's and low-speed connections, but it is not so likely. Actually, it will not happen for socket send buffers which are larger than three MSS segments. Furthermore, for small MSS's the discrepancy between the normal and degraded throughput is not that big, which makes it difficult to detect or uninteresting to investigate into. We know of one paper reporting on performance degradation due to the interaction between the delayed acknowledgment strategy and Nagle's algorithm, namely Crowcroft et al [7]. They report on a boundary effect of remote procedure call (RPC) response time over TCP and ethernet.

The most straightforward way to *prevent* many of the deadlock situations is to switch off Nagle's algorithm. As will be presented, there is hardly any performance penalty having it switched off. A straightforward *avoidance* solution is to ensure that the send socket buffer is equal or greater than the receive socket buffer or than three MSS's. These and other alternatives which require small changes in the TCP implementation will be discussed.

This paper explains what causes the deadlocks and discusses some alternatives to solve the underlying problems. As many of today's TCP implementations incorporate many of the same SunOS optimization mechanisms, we feel that this should be of interest to a broader audience than the SunOS users. The rest of this paper is outlined as follows. Section two summarizes the socket layer, the TCP protocol, and operating system and implementation issues of importance to understand the protocol behavior. Section three goes into a detailed description of the cause

of throughput degradation. Section four discusses possible solutions. Section five contains conclusions. The reader with experience on TCP implementations in Unix BSD environments may want to skip the next section.

2. TCP IN A BSD-BASED UNIX ENVIRONMENT

In most Unix systems, the transport and lower layer protocols are implemented as part of the kernel. There are several reasons for doing this, see for example [8] [9]. The user data to be transmitted is located in user space. On a write system call, user data in the write call is copied from application address space to kernel address space so that TCP and other protocols can do the further processing of the data. Similarly, on reception the requested amount of user data is copied from kernel address space to application space.

TCP is a bidirectional protocol. It establishes a connection and each peer informs the other about its current window size. The window size refers to the number of bytes rather than the number of packets. The sender sends segments which could be as small as a single byte and as large as 64 kbytes. However, TCP sets an MSS per connection. It is normally set to the network MTU minus the size of the TCP/IP header [5]. The MTU is 1500 bytes for ethernet and 9244 bytes for our FORE ATM SBA-100 driver version 2.0. The normal TCP/IP header is 40 bytes which means that the MSS for the measurements in this paper is 9204 bytes.

2.1. Network memory management in the socket layer

In BSD based systems, as for instance SunOS 4.1.1, the socket layer acts as the interface between the application in user space and the protocols within the kernel. The socket layer offers an application programming interface, i.e. system calls such as write and read. A system provided identifier is used in the system calls in order to identify different connections with application processes. Associated with this identifier are two socket data buffers, one for data to the kernel (write) and one for data to the application (read). Each socket buffer consists of

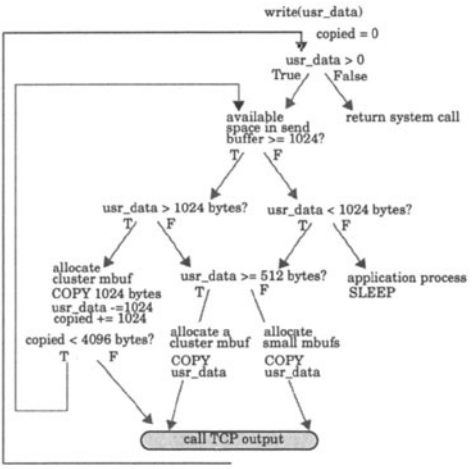


Figure 1. Transmit socket layer data copy strategy

an ordered chain of mbufs [10]. An mbuf is a data structure used by all kernel protocols in SunOS 4.x and by many other BSD systems as well. Data to and from the application is copied to and from these mbuf chains. Associated with the socket identifier is a variable which holds the number of used mbufs in each direction and a variable for the current amount of bytes in these chains of mbufs. The user can set a maximum allowed number of bytes in these chains by using the SO_RCVBUF and SO_SNDBUF socket options.

The application process is put to "sleep" if the socket layer is unable to copy all the application data in the write system call into the buffer. For further progress it has to wait until buffer space is released.

There are two types of mbufs, the "small" mbufs which hold 112 bytes of data and "cluster" mbufs which can take 1 kbyte of data [10]. Whenever possible, the system tries to use cluster mbufs when copying data from user address space to the socket mbuf chain roughly as illustrated in Figure 1. Use of cluster mbufs means that the system can avoid copy operations within the kernel by using a pointer and a reference count instead. Note in Figure 1 that the TCP protocol output routine is called either after all data in the write system call has been copied or after 4096 bytes of data have been copied, whatever occurs first. The motive for copying 4096 bytes into the socket send buffer is to exploit parallelism between the kernel protocol execution and the network interface packet transmission. However, this is only true for networks with a maximum transmission unit (MTU) smaller than 4096 bytes.

2.2. TCP acknowledgment strategy and flow control

TCP's end-to-end flow control is through a sliding window mechanism where the receiver announces its free buffer space to the transmitter. Therefore, when data is copied to the application the receiver checks if a window update packet should be returned. The algorithm for sending a window update [10] is roughly depicted in Figure 2. An update is sent if the window can slide more than either a) 35% of the receive buffer size or b) two MSS segments.

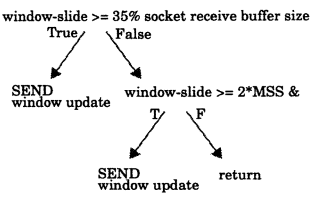


Figure 2. TCP window update algorithm

Both the socket send and receive buffers limit the amount of data that can be outstanding between two communicating TCP peers. The available space (maximum - current amount of data bytes) of the socket receive buffer is used to set the announced TCP window size to ensure that the sender will not send more data than can be received. The send socket buffer is used as the repository for TCP segments in case of retransmissions. Since data bytes remain in the send socket buffer until they are acknowledged, the available space for copying in new data into the socket send buffer is further limited.

The receiving peer sends acknowledgments to the sending peer. In case of lost segments, a time-out function at the sender will generate a retransmission of the unacknowledged bytes. Acknowledgments are delayed until they can be piggybacked onto either data segments in the

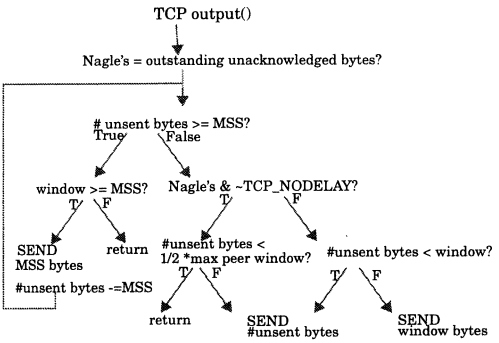


Figure 3. TCP segmentation of the send socket buffer

reverse direction or on window update packets [11]. (A compile option can set TCP to acknowledge each incoming segment.) In addition, explicit acknowledgments are cyclically generated every 200 ms. These 200 ms spaced timer generated acknowledgments are independent of the point of time of connection set-up or the last (not necessarily final) segment reception on this connection.

2.3. Nagle's algorithm

Nagle's algorithm [12] was introduced as a solution to the "small-packet problem". It was observed that TCP sent many small segments which resulted in unnecessary header and processing overhead. Nagle's algorithm inhibits sending TCP segments which are smaller than the assigned MSS if any previously transmitted data on the connection remains unacknowledged, see Figure 3. Nagle's algorithm can be switched off by the TCP_NODELAY option. Switching off Nagle's algorithm is necessary for applications which send small amount of data with no replies, such as a stream of mouse events which has no data in the reverse direction. Still, Nagle's algorithm is recommended on both telnet and ftp connections, and the default TCP configuration uses Nagle's algorithm.

According to Figure 3, a "small" segment is less than MSS bytes. A reflection is that in high-speed networks with large MTUs, Nagle's "small" segments are not actually small anymore. For our ATM network a "small" segment is less than 9204 bytes.

2.4. Invocation of TCP routines

On transmit the TCP output routine is initiated by the write system call. In Unix, a process calling the kernel is never preempted by another process while executing a system call [10]. The kernel call must explicitly give up the processor by a sleep call or run to completion of the system call. System calls appear *synchronously* to the application, i.e. the application process is blocked until the system call returns. The time until return includes any sleep calls while in the kernel. This means that a write system call with large user data sizes is blocked until all data are processed by both the socket and the protocol layers. As can be seen in Figure 1 the process does a sleep when there is not enough space in the socket buffer for all the application data.

System call execution might however be interrupted by the bottom half of the kernel, by hardware interrupts. They occur *asynchronously* and unrelated to the current system call processing.

The receiving application does a read system call which is blocked until there is something to read from the socket layer. On a frame arrival the network interface will receive the frame and generate a hardware interrupt. The hardware interrupt routine runs the device driver, copies data to mbufs, and thereafter initiates a software interrupt. This interrupt handling routine calls the higher-layer protocol e.g. IP which calls the TCP input routine. After TCP has processed data and put it into the receive socket buffer a wake-up call is executed which puts the application process back on the scheduling queue.

An incoming segment with window update and/or acknowledgment information may trigger new data segments to be sent in the other direction. This depends on the current number of bytes in the socket send buffer, Nagle's algorithm and the size of the announced window as previously described. The initiation of segment transfer(s) on acknowledgment reception is one of the points which causes the deadlocks. It may be anticipated that the send algorithm should strive to form as large segments as possible in order to reduce overhead. The obvious thing to do would be to copy as much as possible into the buffer before TCP output is called, especially since an acknowledgment will release buffer space. This is not the case. On the contrary, the action is to first transmit available bytes and then to ask for a refill of the buffer. The argument for doing it in this order is that the application process must be woken up and put into the scheduler queue in order to copy more data. This could cause an unacceptable delay. As a consequence TCP may send small segments. As will be shown later, this order has a more devastating consequence when Nagle's algorithm decides not to send a non-MSS segment.

3. OBSERVED TCP THROUGHPUT DEADLOCKS

All performance measurements in this paper are run by letting TCP transfer 16 Mbytes memory-to-memory between two Sun IPXs using the FORE ASX-111 ATM switch and FORE ATM SBA-100/175 cards (140 Mbit/s physical transmission) with the 2.0 device driver. The user data size of the write/read system call was 8192 bytes. The ATM network interface MTU is 9244 bytes, making TCP compute its MSS to 9204 bytes. Each reported throughput measure is an average of 25 runs. Table 1 presents the throughput for different sizes of the socket send and receive buffers.

In the following we will describe throughput degradations depicted as grey shaded entries in Table 1. S is the size of the send socket buffer. R is the size of the receive socket buffer. As can be seen from the table, dramatic drops in throughput occur when the receive buffer space is equal or less than the sender space. The slow-start [13] behavior is not an issue in these TCP performance measurements, since both the sender and receiver reside on the same IP subnetwork. The degradations in Table 1 are instead due to either the inherent delayed acknowledgment strategy, a combination of the delayed acknowledgment strategy and use of Nagle's algorithm, the sender-side silly-window avoidance rule, or cell loss in the ATM receive interface.

3.1. Classification of the throughput anomalies

In this section we will classify the shaded areas in Table 1. The degradation caused by cell loss or the silly-window avoidance effect will not be discussed in detail. The cell loss happens for large TCP windows which results in buffer overflow at the receiving ATM interface. A cell loss will cause a TCP segment retransmission. The sender-side silly-window syndrome avoidance [10] may occur when the send socket buffer size is more than MSS bytes larger than the receive socket buffer size. We will focus on the other classes.

Table 1 TCP Mbit/s throughput on an ATM network for combinations of socket buffer sizes and a user buffer size of 8192 bytes

S \ R	4k	8k	16k	24k	32k	40k	48k	52k*
4k	11.58	11.78	0.16	0.16	0.16	0.16	0.16	0.16
8k	13.31	15.02	0.16	0.16	0.16	0.16	0.16	0.16
16k	13.47	16.39	18.67	0.34	0.34	0.33	0.49	0.49
24k	13.68	16.46	19.67	19.07	19.98	19.69	1.85	6.63
32k	13.67	16.52	16.88	19.93	21.16	20.91	20.11	20.31
40k	13.07	16.33	17.02	20.02	21.12	21.01	20.58	20.66
48k	13.74	16.19	17.06	19.95	21.20	20.98	6.06	5.39
52k*	13.59	16.08	16.83	20.02	21.28	21.08	6.86	6.78

*52428 bytes, max SunOS socket buffer size

Predictable from the acknowledgment strategy

Combination of socket copy rule and Nagle's algorithm

Combination of timer acknowledgment and Nagle's algorithm

Cell loss -> packet loss

Sender-side silly-window syndrom avoidence

For the grey entries in Table 1 the throughput drop is caused by the same phenomenon, the sender cannot transmit enough data to trigger a window update and a piggybacked acknowledgment from the receiver. We have a deadlock situation where the sender refrains from sending more data, and the receiver refrains from returning an acknowledgment. This deadlock can only be resolved by the cyclically timer generated acknowledgment. Thereafter, the sender starts to send again, but after a while the connection will be back in the same deadlocked situation. Hence, the connection has a stop-and-go behavior of which data is prompted by the 200 ms cyclically generated acknowledgment. The *deadlock* throughput is decided by the cycle for timer generated acknowledgments and the amount of data transmitted until the next deadlock. (The difference in deadlock throughput is only due to the size and the number of the segments transmitted in-between connection deadlocks.)

The combinations of send, S, and receive, R, socket buffer sizes which cause these deadlock situations are marked with two shades of grey in Figure 4. For combinations in the darker area, the connection goes directly into the deadlock situation. For the lighter grey area it may take some time before it happens. It takes longer time for the S=24k R=48k/52k entries in Table 1 compared to the other entries. Due to the measure method we therefore get a higher average throughput than the deadlock throughput for these entries. When the connection gets into the deadlock situation, the throughput is 0.67 Mbit/s.

The throughput deadlocks can be partitioned into three classes depending on the reason for causing the deadlock:

- Deadlocks predictable from the window update rules
- Deadlocks caused by the socket layer data copying rules and Nagle's algorithm
- Deadlocks caused by the timer generated acknowledgment and Nagle's algorithm

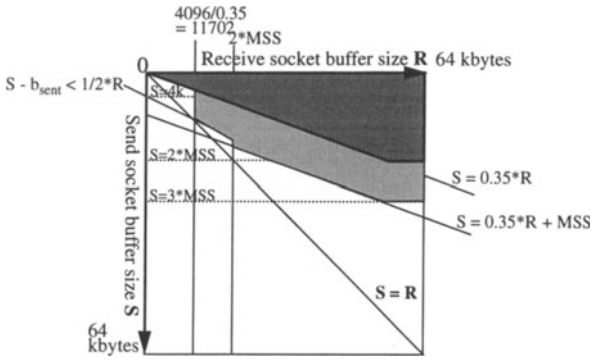


Figure 4. Anomalous socket buffer size combinations for MSS=9204 bytes

3.1.1. Deadlocks predictable from the window update rules

In this section we will discuss the dark grey area in Figure 4. In this class, the send socket buffer size is less than 35% of the receive socket buffer size, and also less than twice the MSS of the ATM network. Knowing the window update algorithm and acknowledgment strategy of TCP, these results are predictable. Even if the whole send socket buffer is sent, it is not enough to trigger a window update onto which to piggyback an acknowledgment.

The following inequalities hold between the send socket buffer and the receive socket buffer marked with the dark grey area in Figure 4:

$$(S < 35\%R) \;\&\; (S < 2*MSS) \tag{1}$$

For example, consider the entry S=8k R=24k in Table 1, which yields 0.16 Mbit/s. The socket layer copies 4 kbytes into the socket send buffer before it calls TCP. At the receiving side, these 4 kbytes are less than 35% of 24k and less than 2*MSS. Therefore, no window update will be returned. The sender can, and will copy another 4 kbytes into the send socket buffer, but due to Nagle's algorithm the sender refrains from sending these 4 kbytes until an acknowledgment has been received. There is now no more space for copying in additional data. The connection is deadlocked and the returned acknowledgment will be timer generated. When this acknowledgment is received by the sender, the sender first transmits the remaining bytes in the send socket buffer before it starts copying more data from the application to the socket buffer. Thus, the connection is stop-and-go with 4 kbytes sent every 200 ms which gives a throughput of 0.16 Mbit/s. Actually, the connection will deadlock independent of Nagle's algorithm. Even if the whole 8 kbyte send socket buffer is sent, it will still be less than 35% of the receive buffer.

Now consider the entries S=16k R=48k/52k which yield 0.49 Mbit/s. Here 3084 bytes + 9204 bytes are immediately sent as two segments when the timer generated acknowledgment releases buffer space. Otherwise the behavior is as described above.

3.1.2. Deadlocks caused by the data copying rules and Nagle's algorithm

In this class S is larger than 35%R or 2*MSS. The deadlock situations are therefore not predictable according to the TCP window update strategy. Anyhow, the TCP connection sooner

or later phases into a behavior which relies on timer generated acknowledgments to resolve deadlocks.

The light grey area in Figure 4 is bounded by the dark grey area and the following inequalities between the send socket buffer, S, and the receive socket buffer, R:

$$(S < 35\%R+MSS) \;\&\; (S < 3*MSS) \tag{2}$$

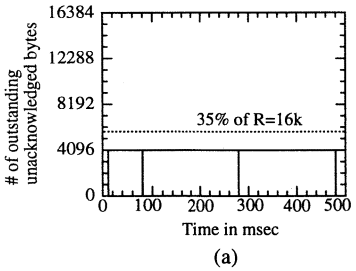
For small S and R there are some boundary effects, which will be discussed later. The upper limit 3*MSS is caused by the implementation decision to first send available bytes in the socket send buffer and thereafter copy from user space. If it were done the other way round, the upper limit would instead be 2*MSS .

For example, consider the entry S=8k R=16k which yields 0.16 Mbit/s. S is big enough (50% of R) to trigger the 35% window update rule. This is what happens; The first write system call of 8 kbytes results in only 4 kbytes being copied into the send socket buffer before TCP is called, see Figure 1. Figure 5 (a) illustrates this by showing for the S=8k R=16k entry, the sender side data segment transmission and acknowledgment reception, that is, the number of outstanding unacknowledged bytes. TCP transmits a segment with 4096 bytes, and the last 4096 bytes of the write call are copied into the send socket buffer. At this point in time there are 4k unacknowledged bytes, and 4k new unsent bytes in the send buffer. Due to Nagle's algorithm these new 4 kbytes can not be transmitted since they are less than MSS. At the receiver the window can slide only 25% (4k/16k) so there is no window update to piggyback an acknowledgment onto. At this stage, the connection is deadlocked and TCP acts as a stop-and-go protocol with a window of 4096 bytes, and acknowledgments generated every 200 ms. The achieved throughput is 20 kbyte/s or 0.16 Mbit/s. After a small initial phase the entry S=16k R=40k also goes directly into deadlock and transmits 8 kbytes in-between deadlocks.

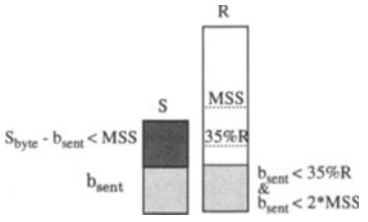
Common to these two examples is that the connection immediately gets into deadlock. A deadlock situation occurs when b_{sent} bytes are sent which are not enough to advance the window but are enough to block the sender to create a new MSS segment. Connection deadlock happens when

$$(b_{sent} < 2*MSS) \;\&\; (b_{sent} < 35\%R) \;\&\; ((S_{byte} - b_{sent}) < MSS) \tag{3}$$

S_{byte} is the number of data bytes in the socket send buffer. The inequality is illustrated in Figure 5 (b).



(a)



(b)

Figure 5. (a) Traces of b_{sent}, S=8k R=16k
(b) Snapshot of bytes in send and receive socket buffers at deadlock

The sender may send several segments in-between deadlocks, thus the throughput may vary. The absolute upper throughput limit after deadlock is though (35%R+MSS)/200ms.

3.1.3. Deadlock caused by the timer generated acknowledgment and Nagle's algorithm

For entries like S=16k R= 24k/32k it seems that there should be no problem because after 4 kbytes are transmitted, a full MSS of 9204 bytes can be constructed, which should prompt a window update. But assume now instead that at some point in time there will be, say, an 8 kbyte segment sent. This 8 kbyte segment is not big enough to advance the window and the available space in send socket buffer is not big enough to create a full MSS. But how could there be an 8k segment sent? In short, the timer generated acknowledgment may arrive just after that 8k has been copied into an empty send buffer. When this situation is reached, 8k is sent every 200 ms. Observe that this is caused by the fact that the implementation on reception of an acknowledgment first sends what is available in the send buffer and thereafter copies more bytes into the buffer. Appendix A gives a detailed presentation of the 0.34 Mbit/s throughput result with a 16k send buffer and a 32k receive buffer.

Figure 6 (a) depicts how a S=16k R=32k connection gets into deadlock after about 350 ms. Assume the sender has transmitted b_{sent} bytes. A timer generated acknowledgment which acknowledges b_{ack} of these b_{sent} bytes such that (b_{sent} - b_{ack}) satisfies (3), that is

$$((b_{sent} - b_{ack}) < 35\%R) \;\&\; ((b_{sent} - b_{ack}) < 2*MSS) \;\&\; ((S_{byte} - (b_{sent} - b_{ack})) < MSS) \qquad (4)$$

is the reason for the deadlock. Figure 6 (b) presents a snapshot of the socket send buffer in this situation. Due to the segment flow on the TCP connection, the probability of a timer generated acknowledgment to actually acknowledge b_{ack} bytes as above is very high; The connection deadlocks within 600 ms. See Appendix A for details.

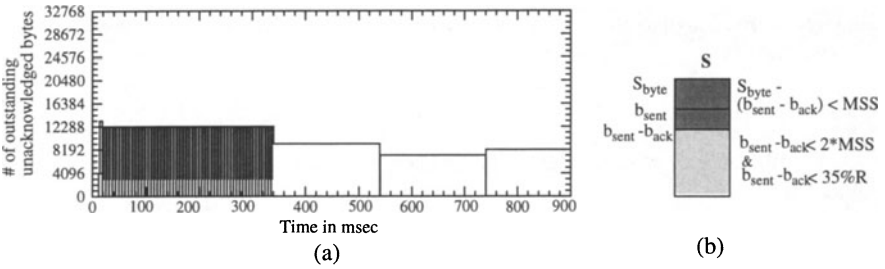


Figure 6. (a) Trace of a S=16k R=32k connection (b) Snapshot of the socket send buffer

3.1.4. Boundary effects

With the SunOS 4.1.x socket layer optimization of calling the protocol for at least every 4 kbytes results in the first segment being maximum 4 kbytes, independent of the user data in the write call as long as this is larger than 4 kbytes. This will get the connection directly into deadlock if R is larger than 4096/35% = 11702 bytes. This gives a vertical line at 4096/0.35 = 11702 bytes in Figure 4. If R < 11702 the deadlock depends on the user data size. A smaller user data size than 4k will move this boundary to the left.

Another boundary effect is as follows. When b_{sent} bytes are sent, there is potential for

another S_{byte} - b_{sent} bytes to be transmitted. Depending on b_{sent} and S_{byte}, this segment may be less than MSS, and with Nagle's algorithm in use, according to Figure 3 such a small segment is sent only if

$$(S_{byte} - b_{sent} > 1/2*R) \quad (5)$$

Inequality (5) is due to TCP sending a segment if the size of the segment is at least half the maximum advertized receive window. This was done to cope with an initial problem of the sender avoidance of the silly-window syndrome when communicating with hosts with tiny buffers, e.g. 512 bytes [10]. Thus, if S_{byte} - b_{sent} is larger than half the maximum advertized window, R, for R less than 2*MSS a small segment is transmitted independent of Nagle's algorithm. In Figure 4 (5) is drawn with b_{sent} equal to 4 kbytes.

3.1.5. Deadlocks with small MSS

The deadlocks above occur also on other networks. The larger the network interface MTU, the larger the hazardous send and receive socket size combination areas. The hazardous socket size combinations for ethernet is (without the boundary effects) depicted in Figure 7. Due to the smaller MTU the number of combinations to avoid is much smaller.

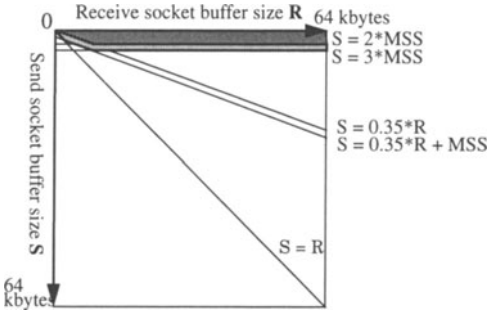


Figure 7. Ethernet anomalous socket buffer size combinations, MSS = 1460 bytes

4. DEFEATING THE DEADLOCKS

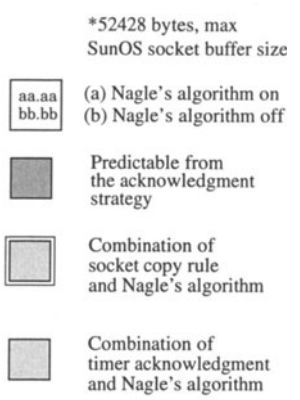
This section discusses how to avoid or prevent the deadlock situations. The obvious avoidance solution is to keep away from the dangerous S and R combinations, i.e. to ensure that S >= 3*MSS or S >= R. Another straightforward prevention solution is to turn off Nagle's algorithm. Other alternatives are to change Nagle's algorithm, the size of MSS, and to remove the 4k limit on the number of bytes copied to the socket buffer before TCP is called.

4.1. Preventing deadlocks by turning Nagle's algorithm off

Setting the TCP_NODELAY option removes the lightly shaded low throughput entries without a significant performance penalty for other socket size combinations, see Table 2 for pairwise comparison of the upper (Nagle's on) and lower (Nagle's off) rows of each entry. As expected, the throughput performance in the darkly shaded area is still doomed to be low. In some of the darkly shaded entries the throughput increase is due to the fact that the sender

Table 2 TCP Mbit/s throughput over ATM
(a) with Nagle's algorithm (b) without Nagle's algorithm

S \ R	4k	8k	16k	24k	32k	40k	48k	52k*
4k	11.58 11.77	11.78 11.78	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16
8k	13.31 13.31	15.02 15.13	0.16 15.30	0.16 0.33	0.16 0.33	0.16 0.33	0.16 0.33	0.16 0.33
16k	13.47 13.47	16.39 16.23	18.67 18.76	0.34 18.98	0.34 19.31	0.33 19.10	0.49 0.66	0.49 0.66
24k	13.68 13.68	16.46 16.13	19.67 19.77	19.07 20.59	19.98 20.29	19.69 20.33	1.85 19.28	6.63 19.26
32k	13.67 13.68	16.52 15.88	16.88 16.42	19.93 19.74	21.16 20.98	20.91 20.92	20.11 19.86	20.31 19.84
40k	13.07 13.44	16.33 16.27	17.02 16.91	20.02 19.87	21.12 20.94	21.01 20.89	20.58 20.45	20.66 20.70
48k	13.74 13.07	16.19 16.10	17.06 16.47	19.95 19.53	21.20 20.92	20.98 20.80	6.06 17.53	5.39 17.17
52k*	13.59 13.75	16.08 15.65	16.83 16.76	20.02 19.78	21.28 21.21	21.08 20.73	6.86 14.84	6.78 14.39



transmits the whole send buffer size in-between deadlocks. The throughput degradation due to packet loss is reduced, since the receiver copes better with a more even distribution of (smaller) packets in time.

4.2. Incorporating smaller changes in TCP

To cope with the deadlock due to the timer generated acknowledgment and Nagle's algorithm requires changes to current implementations. We identify two possible changes which we think have little impact on TCP and existing implementations:

- A change in Nagle's algorithm in order to allow small packets to be transmitted if there are more than MSS outstanding unacknowledged bytes. This is a small adjustment which implies expanding Nagle's statement with a conditional test on the *number* of unacknowledged bytes.
- Set the MSS to a power-of-two multiple of the 1024 byte cluster page size. Protocol and network memory management optimizations are thereby performed on corresponding chunk sizes.

In the following we present performance results after the changes above.

4.2.1. A change in Nagle's algorithm

By changing Nagle's algorithm to allow small segments we will significantly reduce the probability of entering deadlock situations. As can be seen from the upper line of each of the entries in Table 3, this change to Nagle's algorithm significantly improves the throughput performance of the S=16k R=24k/32k and S=24k R=48k/52k entries. These are the cases of which the timer generated acknowledgment gets the connection into an anomalous behavior.

Table 3 TCP throughput over ATM in Mbit/s,
(a) with a change to Nagle's algorithm (b) TCP MSS of 8192 bytes

S \ R	4k	8k	16k	24k	32k	40k	48k	52k*
4k	11.94 12.21	11.88 11.72	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16
8k	13.08 13.47	15.24 15.01	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16	0.16 0.16
16k	13.14 13.74	16.35 16.72	18.93 18.95	13.01 17.80	11.16 17.83	0.49 0.49	0.49 0.49	0.49 0.49
24k	13.17 13.76	16.35 16.50	19.01 19.08	19.61 20.80	20.24 20.55	20.27 20.46	20.30 19.14	20.32 19.06
32k	13.16 13.75	16.21 16.46	17.07 20.56	19.92 21.10	21.21 21.14	20.91 21.02	20.92 20.59	20.69 20.62
40k	13.19 13.76	16.30 16.02	16.78 20.19	19.92 20.86	21.19 21.22	20.96 21.07	20.67 20.87	20.57 21.02
48k	13.21 13.80	16.57 16.86	17.41 21.12	20.56 21.39	21.53 21.29	20.61 21.08	15.96 16.43	16.48 15.23
52k*	13.07 13.74	16.76 16.83	17.77 21.17	20.66 21.47	21.37 21.24	20.97 21.04	12.89 19.94	12.12 18.97

*52428 bytes, max SunOS socket buffer size

aa.aa / bb.bb — Nagle's change TCP MSS 8 kbytes

Predictable from the acknowledgment strategy

Combination of socket copy rule and Nagle's algorithm

The lack of improvement of the S=8k R=16k and S=16k R=40k entries is due to the socket layer 4k copy rule. The average throughput reported in the S=16k R=24k/32k entries is improved since it now takes longer time to get into the deadlock.

4.2.2. A 8192 byte TCP MSS

A TCP MSS of 8192 bytes can be achieved by either TCP rounding the computed MSS to its nearest lower integer multiple of 1024 bytes, or by setting the ATM MTU to 8232 bytes. In the first case the TCP code must be changed. The second case requires a change in the ATM network driver. Setting the ATM MTU to 8232 bytes makes TCP compute its MSS as 40 bytes less, that is 8192 bytes.

The bottom line of each of the entries in Table 3 present the throughput results with the TCP MSS of 8192 bytes. The S=16k R=48k/52k entries are now classified as medium grey, since S now is big enough to hold two MSS segments. The throughput of these entries and the S=8k R=16k and S=16k R=40k entries remain low because it is the socket layer copy optimization which causes this low throughput. The S=16k R=24k/32k and S=24k R=48k/52k entries have improved significantly. Note that as a side effect an MSS of 8192 bytes improves the throughput for the sender-side silly-window-avoidance entries, and the packet loss entries.

4.3. No explicit 4k limit in the socket copy rules

A change to the socket layer removes the explicit limit of 4 kbytes copied to the socket buffer before the protocol is called. The inevitable limit is now only the available space in the socket buffer. This change makes the possibility of deadlock more dependent on the user data size, but does not prevent the deadlocks. With a user data size of 8 kbytes as in the previous measurements this change avoids only the S=8k R=16k deadlock. On the contrary, with a user data size of 4 kbytes the connection would deadlock as described earlier.

5. CONCLUSIONS

In this paper we have pointed out common socket buffer size settings which force TCP into deadlock situations over high-speed networks with large MTUs. The number of socket size combinations increases with an increasing MTU relative to the send and receive socket buffer sizes. The window update rules and Nagle's definition of a "small" segment are of importance. On high-speed networks with large transmission units, the unit which Nagle's algorithm considers small is not small anymore.

The predictable deadlocks are due to the TCP window update rules, the other deadlock situations are due to:

- A bad interaction between Nagle's algorithm and the delayed acknowledgment strategy
- The socket layer copy rule (apparently optimized for ethernet)
- The action sequence of the TCP implementation on acknowledgment reception
- The explicit timer generated acknowledgment which operates independent from the rest of the protocol

We presented ways of getting round the throughput deadlocks. The conclusions from these are

- A send socket buffer equal or larger than three MSS's or equal or larger than the receive socket buffer avoids all deadlocks.
- Turning off Nagle's algorithm prevents non-predictable deadlocks.
- A change to Nagle's algorithm to allow small segments to be transmitted if there are at least MSS unacknowledged bytes, greatly reduced the probability of connection deadlock.
- Setting the MSS to a power-of-two multiple of 1 kbyte causes segment sizes to be more in line with the internal network memory management rules and common socket buffer sizes. To avoid some of the deadlocks an appropriate TCP [1] MSS size is 8k.

REFERENCES

1. Postel, J. Transmission Control Protocol, protocol specification. RFC 793, September 1981.
2. Comer, D. Internetworking with TCP/IP, principles, protocols, and architecture. Englewood Cliffs, NJ, Prentice-Hall. ISBN 0-13-470188-7. 1988.
3. V. Jacobsen, B. Braden, and D.Borman "TCP extensions for high-performance". RFC 1323, May 1992.
4. Partridge, C. Gigabit networking. Addison-Wesley. ISBN 0-201-56333-9. 1993.
5. Postel, J. The TCP maximum segment size and related topics. RFC 879, November 1983.
6. Braden, R (ed). Requirements for Internet hosts - communication layers. RFC 1122, October 1989.
7. Crowcroft, J, Wakeman, I, Wang, Z, and Sirovica, D. Is layering harmful? IEEE Network, 6 (1), 20-24, January 1992.
8. Clark, D D. Modularity and efficiency in protocol implementation. RFC 817, July 1982.
9. Mogul, J C, Rashid R F, and Accetta, MJ. The packet filter: an efficient mechanism for user-level network code. Proc. of ACM SOSP, 1987, 39 - 51.

10. Leffler, S J et al. 4.3 BSD Unix operating system. Reading, Mass., Addison-Wesley. ISBN 0-201-06196-1. 1989.
11. Clark, D D. Window and acknowledgment strategy in TCP. RFC 813, July 1988.
12. Nagle, J. Congestion control in TCP/IP internetworks. RFC 896, January 1984.
13. Jacobsen, V. 1988. Congestion avoidence and control. Proc. of ACM SIGCOMM'88, 314-329, Palo Alto, USA, 16-19 August 1988.

APPENDIX

A Throughput with a 16k send and a 32k receive socket buffer

In the following we will describe in detail how a throughput deadlock arises with a socket send buffer of 16 and a socket receive buffer of 32 kbytes. The diagram in Table A.1 is used to illustrate TCP internal actions and state and the packet flow on the network.

The sender is a loop of write(8k) calls, while the receiver is a loop of read(8k) calls. The data segments are represented with $DATA_X$ where X is the number of user bytes in the packets, and acknowledgments with ACK_Y where Y is the number of bytes the packets acknowledge.

The data transfer phase starts with a write(8k). The socket layer copies 4 kbytes into the socket buffer before it calls the TCP protocol through tcp_output(). A packet with 4 kbyte user data is transmitted. After the return from the tcp_output() routine, the socket layer continues by copying more bytes from the current and next write(8k) call until the socket buffer is full. At this stage, the call produces a segment of length 9204 (MSS) bytes. 13300 bytes have now been transmitted, but are not acknowledged. The last 3084 bytes in the send socket buffer are not transmitted since they are less than MSS (line 7).

After the segments have arrived at the receiver, the application reads them in two chunks. After the second read(), the window can slide 13300/32768 = 40.6%, and a window update with an acknowledgment is returned. The acknowledgment releases 13300 bytes in the send socket buffer and acknowledges all outstanding bytes. When the window update is received, TCP first sends the remaining 3084 data bytes in the send socket buffer before the application is scheduled to run. Thereafter, another segment of MSS bytes will be transmitted. Due to the generated segment size pattern on the sender side, an acknowledgment will be returned immediately after having received a 9204 byte segment, since the window will slide more than 35%.

The sequence of sending a 3084 byte and a 9204 byte segment followed by an acknowledgment is repeated until the sender receives a timer generated acknowledgment, TO. Such an acknowledgment is generated on line 21, and it arrives at the sender on line 25 and acknowledges less than MSS bytes.

The timer generated acknowledgment on line 21 releases only 3084 bytes in the socket send buffer when it is received on line 25. Thus, in the socket send buffer, 9204 bytes still remain unacknowledged. The 3084 bytes ready to be transmitted on line 25 are not sent due to Nagle's algorithm. The reception of the acknowledgment wakes up the application. Another 4 kbytes can be copied into the send buffer which now contains 16 kbytes. There are 7180 (16384-9204) bytes left in the socket send buffer ready to be transmitted. Due to Nagle's algorithm, no segment is transmitted because there are outstanding unacknowledged bytes.

On line 28 a new 200 ms spaced timer generated acknowledgment is returned. It acknowledges 9204 bytes and at this time the sender does not have any outstanding unacknowledged bytes. Therefore, the remaining 7180 bytes in the send socket buffer are

Table A.1 ATM 16k send and 32k receive socket buffer, user data size of 8 kbytes

	TRANSMIT 8192 bytes in each system call	Socket copy before TCP call	From TCP to proc in sleep	Bytes in socket send buffer	Number of unacked bytes	PACKETS on the wire	Bytes in socket receive buffer	From TCP to proc in sleep + TO	RECEIVE maximum 8192 bytes in each system call
1	write(8k)	4096		4096	4096	>> $DATA_{4096}$ >>	4096		
2		4096		8192	4096		4096		
3	write(8k)	4096		12288	4096		4096		
4		4096		16384	13300	>> $DATA_{9204}$ >>	13300	wkup	
5	write(8k)	sleep			13300		5180		read(8192)
6			wkup	3084	0	<< ACK_{13300} <<	0		read(5108)sleep
7				3084	3084	>> $DATA_{3084}$ >>	3084	wkup	
8		4096		7180	3084		0		read(3084)sleep
9		4096		11276	3084		0		
10	write(8k)	4096		15372	12288	>> $DATA_{9204}$ >>	9204	wkup	
11		sleep		15372	12288		1012		read(8192)
12			wkup	3084	0	<< ACK_{12288} <<	0		read(1012)
13				3084	3084	>> $DATA_{3084}$ >>	3084	wkup	
14		4096		7180	3084		0		read(3084)
15	write(8k)	4096		11276	3084		0		
16		4096		15372	12288	>> $DATA_{9204}$ >>	9204		
17	write(8k)	sleep		15372	12288		1012		read(8192))
18			wkup	3084	0	<< ACK_{12288} <<	0		read(1012)
19				3084	3084	>> $DATA_{3084}$ >>	3084		
20				3084	3084		0		read(3084)sleep
21		4096		7180	3084	ACK_{3084} <<	0	TO	
22		4096		11276	3084		0		
23	write(8k)	4096		15372	12288	>> $DATA_{9204}$ >>	9204	wkup	
24		sleep		15372	12288		9204		read(8192)
25			wkup	12288	9204	<< (ACK_{3084})	0		read(1012)sleep
26		4096		16384	9204		0		
27	write(8k)	sleep		16384	9204		0		
28			wkup	7180	0	<< ACK_{9204} <<	0	TO	
29				7180	7180	>> $DATA_{7180}$ >>	7180	wkup	
30		4096		11276	7180		0		read(7180)sleep
31		4096		15372	7180		0		
32	write(8k)	sleep		15372	7180		0		
33			wkup	8192	0	<< ACK_{7180} <<	0	TO	
34				8192	8192	>> $DATA_{8192}$ >>	8192	wkup	
35		4096		12288	8192		0		read(8192)sleep
38		4096		16384	8192		0		
39	write(8k)	sleep		16384	8192		0		
40			wkup	8192	0	<< ACK_{8192} <<	0	TO	
41				8192	8192	>> $DATA_{8192}$ >>	8192	wkup	
42		4096		12288	8192		0		read(8192)sleep
43		4096		16384	8192		0		
44	write(8k)	sleep		16384	8192		0		
45			wkup	8192	0	<< ACK_{8192} <<	0	TO	
46				8192	8192	>> $DATA_{8192}$ >>	8192	wkup	
47		4096		12288	8192		0		read(8192)sleep
48		4096		16384	8192		0		
49			wkup	8192	0	<< ACK_{8192} <<	0	TO	
50				8192	8192	$DATA_{8192}$	4096	wkup	
51				8192	8192		0		read(8192)sleep
52				0	0	ACK_{8192}	0	TO	

transmitted in one segment. At the receiver this segment (line 29) does not trigger a window update. At the sender side, there are now only 7180 bytes in the send socket buffer. Hence, there is space for another 9204 (16384 - 7180) bytes, that is MSS bytes! The acknowledgment on line 28 wakes up the application and twice are 4 kbytes copied into the send socket buffer. A new segment is not transmitted, since its length is less than MSS. The write call on line 32 immediately does a sleep because there is not enough free space in the socket buffer to copy an additional 1024 bytes. This is the case even if there is space for an additional 1012 bytes which would create a full MSS, see Figure 1. At this point in time, there are 15372 bytes in the socket buffer of which 7180 have been transmitted.

The next action is the timer generated acknowledgment on line 33. It acknowledges 7180 bytes. On reception of this acknowledgment at the sender, 8192 bytes are transmitted. Again, no progress is made since the sender can not send an MSS segment. The receiver does not return an acknowledgment as the window can slide only 25%. We have reached the point at which 8192 bytes will be sent every 200 ms. This behavior is repeated (lines 38 thru 52) and gives a throughput of about 40 kbyte/s, or 0.32 Mbit/s.

Figure A.1 presents traces of the data segment transmission and acknowledgment reception on a S=16k R=32k connection. (a) presents the segment transfer until the connection deadlocks, (b) is a magnified vertical slice of the initial segment flow of (a), and (c) presents the timer generated acknowledgment. Taking a closer look on Figure A.1 (b) it is evident that the connection will deadlock with a probability of 1 within 600 ms. The sender transmits the 3084 byte segments approximately 6 ms apart. About 1.7 ms after the transmission of the 3084 byte segment, the 9204 byte segment is transmitted. Within each 200 ms cycle, the time interval the receiver has only 3084 unacknowledged bytes is shifted by approximately 2 ms relative to the time of the timer generated acknowledgment. That is, within the first few 3-5 timer generated acknowledgments the connection is deadlocked.

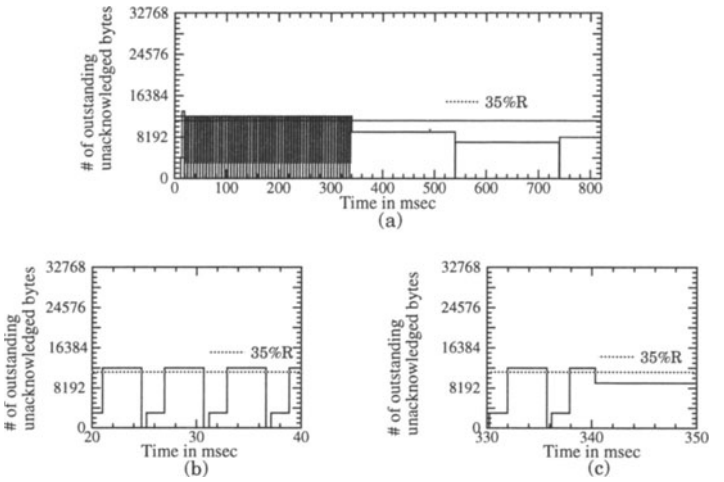


Figure A.1 Trace of data segments and acknowledgments, S=16k R=32k