15

Deadlock-Free TCP Over High-Speed Internet

Rocky K. C. Chang
The Hong Kong Polytechnic University, Kowloon (Hong Kong).

Ho Y. Chan
University of Southern California, Los Angeles, CA, USA.

Adam W. Yeung
Cisco Systems Inc., San Jose, CA, USA.

In this paper, we consider TCP throughput deadlock problems caused by an interplay between the Nagle algorithm, delayed acknowledgment algorithm, and several implementation details. For some combinations of send and receive buffers, a TCP sender cannot send more segments due to the Nagle algorithm and, at the same time, a TCP receiver cannot acknowledge more segments received due to the delayed acknowledgment algorithm. The outcome is a deadlock, which can only be resolved by the receiver's timer. Although the deadlock can take place in any types of networks, it is generally more difficult to ensure deadlock-free connections on high-speed networks. Moreover, the impact is much more significant on high-speed networks, and the deadlock renders the connection practically unusable. Several straightforward solutions, such as turning off the Nagle algorithm and acknowledging every segment, have been proposed; however, they reintroduce the same problems that they were initially designed for. In this paper we propose an adaptive acknowledgment algorithm (A^3) to eliminate throughput deadlocks on the receiver side while preserving the original intent of employing the Nagle algorithm and delayed acknowledgment. An A^3-receiver uses the same delayed acknowledgment as before, but with an additional component to adaptively

compute the acknowledgment threshold, which is adjusted according to the maximum amount of segments sent by the sender. By adapting to the sender's state, an A^3-receiver can avoid deadlocks when there is no network congestion. To further adapt to possible network congestion, A^3-receivers are enhanced by incorporating a slow-start-like algorithm to adjust the acknowledgment threshold when network congestion is suspected. The resulting algorithm is referred to as congestion-sensitive A^3 (CSA^3). Extensive simulation experiments have confirmed the effectiveness of both A^3 and CSA^3.

15.1 INTRODUCTION

Transport control protocol (TCP) continues to dominate Internet traffic by providing end-to-end reliability, flow control, and congestion control services to a number of very popular application and session protocols, such as HTTP, FTP, TELNET, SSL, etc. Being designed as a generic transport protocol back in the mid-1980s, the TCP performance has to keep up with rapid advances in the underlying networking technologies and new application requirements. One area of TCP performance degradation is brought by new data-link technologies' characteristics, such as long delay-bandwidth-product of satellite links [1-3], channel asymmetry of cable modem networks [4], and high error rates of wireless links [5]. Each of these issues affects the growth of TCP sending window size in different ways, but the results are the same: the TCP's throughput is severely limited with respect to the available bandwidth between end hosts.

On the other hand, implementation issues are equally, if not more, important in affecting TCP performance, e.g., [6-8]. The performance of a network protocol realization is particularly important for TCP because TCP is currently defined by implementations rather by formal protocol specifications [7]. In this paper, we consider TCP throughput deadlock problems that are caused mainly by an interplay between the Nagle algorithm, delayed acknowledgment algorithm, and various TCP implementation issues. The Nagle algorithm and delayed acknowledgment algorithm were in fact designed to address the "small-packet problem," which significantly reduces network throughput as a result of sending small-sized data (say 1 byte) in one IP datagram (usually 40 bytes) instead of a full segment. The small-packet problem is also referred to as a silly window syndrome (SWS) problem [9]. The Nagle algorithm is a sender-side SWS avoidance solution, which prevents a sender from sending small segments when there are outstanding segments to be acknowledged [10]. A TCP segment is considered small if it is less than the sender-side maximum segment size (MSS); therefore, a nonMSS-sized segment is considered small. The delayed acknowledgment algorithm, on the other hand, is a receiver-side SWS avoidance solution, which prevents a receiver from acknowledging small segments [11].

A throughput deadlock rises when a pair of TCP sender and receiver gets into a circular-wait situation. That is, the sender cannot send more segments due to

Nagle algorithm, while the receiver cannot send acknowledgments due to the delayed acknowledgment algorithm. Although the deadlock state is finally escaped by the firing of a coarse-spaced delayed acknowledgment timer, the resulting TCP throughput is so low that the connection is practically not usable. A necessary condition for getting into the deadlock situation is that a TCP sender sends nonMSS-sized segments. When the amount of outstanding segments is not enough to trigger an immediate acknowledgment from the receiver, a throughput deadlock will occur. However, there are different factors responsible for the sending of nonMSS-sized segments. For example,

1. An application data may consist of an odd number of MSS-sized segments with a nonMSS-sized final segment. In this case, the receiver may not be able to promptly acknowledge the last MSS-sized segment due to the delayed acknowledgment algorithm. At the same time, the sender is unable to send the nonMSS-sized final segment due to Nagle algorithm. The resulting throughput degradation was reported in the context of persistent HTTP connections [12].
2. Even if the application data size does not fall into scenario (1), nonMSS-sized segments may still be generated as a result of *buffer tearing*, in which an application data is usually broken up into a number of nonMSS-sized segments when copied to the write buffers used by TCP. This is because the write buffer sizes are usually not multiples of MSS.
3. Other OS implementation issues, such as the send-receiver buffer size combinations, data copying rules, and order of actions when receiving acknowledgments, can cause the sender to send nonMSS-sized segments. Throughout degradation as a result of these issues was reported by Moldeklev and Gunningberg [14], and Comer and Lin [15].

The three scenarios discussed above involve the application, socket, and TCP layers. Moreover, all three factors could cause throughput deadlocks on low-speed networks as well as high-speed networks. However, the impact is more noticeable in high-speed, end-to-end connections, e.g., client and server on a 100-Mbps LAN. Furthermore, it is more difficult to guarantee deadlock-free TCP connection in high-speed networks, because the MSS value is usually very high on those networks. Hence, we consider the deadlock problems mainly for high-speed TCP connections. Although the speed of an end-to-end connection is generally not high today, this TCP deadlock problem is expected to have a more significant impact as the effort of deploying TCP/IP on top of many high-speed networks, such as ATM, WDM, and broadband satellites, accelerates.

In terms of resolving throughput deadlocks, Mogul and Minshall [13] proposed an improved implementation of the Nagle algorithm to overcome the first two scenarios of deadlocks. However, they did not address the third scenario. Moldeklev and Gunningberg, on the other hand, proposed several straightforward solutions to solving the deadlock problems arising from the third scenario, such as disabling the Nagle algorithm and acknowledging every segment [14]. These solutions can guarantee deadlock-free connections, but they will clearly reintroduce

the SWS problem. Murayama and Yamaguchi [16] proposed to use new TCP flags to implement "No Delayed ACK" and "Force Delayed ACK" options. But this proposal also requires new TCP implementations on all systems, which is unlikely to happen.

In this paper we propose an adaptive approach to avoid SWS on the receiver side, referred to A^3. An A^3-receiver is the same as a typical TCP receiver except for an additional component that "adaptively" determines when to send acknowledgments based on the information gathered on the sender. In other words, throughput deadlocks are avoided on the receiver, instead of the sender side. The A^3 also does not distinguish the exact causes for throughput deadlocks, and it is therefore designed to handle all three scenarios of deadlocks.

The rest of this paper is organized as follows: In Section 15.2, we first describe necessary background information on the socket buffer management, the Nagle algorithm, and delayed acknowledgment algorithm. We then describe the throughput deadlock problem in more details. In Section 15.3, we introduce the A^3 and show that it eliminates all classes of deadlocks in the absence of network congestion. In Section 15.4, we explain why throughput deadlocks still occur to the A^3 in the presence of network congestion. Subsequently, we introduce Congestion-Sensitive A^3 (CSA^3), and show that the CSA^3 is able to ensure deadlock-free TCP connections even in a congested network environment. Finally, we conclude this paper with future work in Section 15.5.

15.2 THE TCP THROUGHPUT DEADLOCK PROBLEM

15.2.1 Unix Socket Layer

The networking codes in most BSD-based Unix kernels are organized into three layers: socket, protocol, and interface [17]. The socket layer is a protocol-independent interface to the protocol-dependent layer below while the latter two layers implement specific network protocol suites and device drivers for data-link technologies.

Data transfer between the application and protocol layers largely relies on memory buffers. An efficient memory management scheme called *mbufs* (memory buffers) was first introduced in BSD 4.3 and later adopted by SunOS 4.x. The mbufs scheme provides fixed and variable size memory allocation that improves efficiency by reducing physical data replication within the kernel memory space. The mbufs memory is allocated during system initialization and is part of the permanent kernel memory that always resides in the physical memory. There are two types of mbufs: small (or plain) and cluster. The plain mbufs are 128 bytes long with 112 bytes data storage which form a cluster of mbufs when an external page (1024 bytes from mbufs memory pool) is attached to the existing plain mbufs. The cluster mbufs' data is stored exclusively in the external page to facilitate pointer referencing.

In SunOS 4.1.3 bulk data transfer, data will be added in the form of multiple 1024-byte cluster mbufs when the send buffer and user data are larger than or equal to 512 bytes. Figure 15.1 illustrates the data copy routine in the socket layer. One important thing to note from the figure is that as soon as the send buffer collects 4096 bytes of data, it will send them out to the TCP output for delivery, without waiting for more application data. This data copy mechanism, as we will see later, turns out to be an important implementation issue that is partially responsible for TCP throughout deadlock.

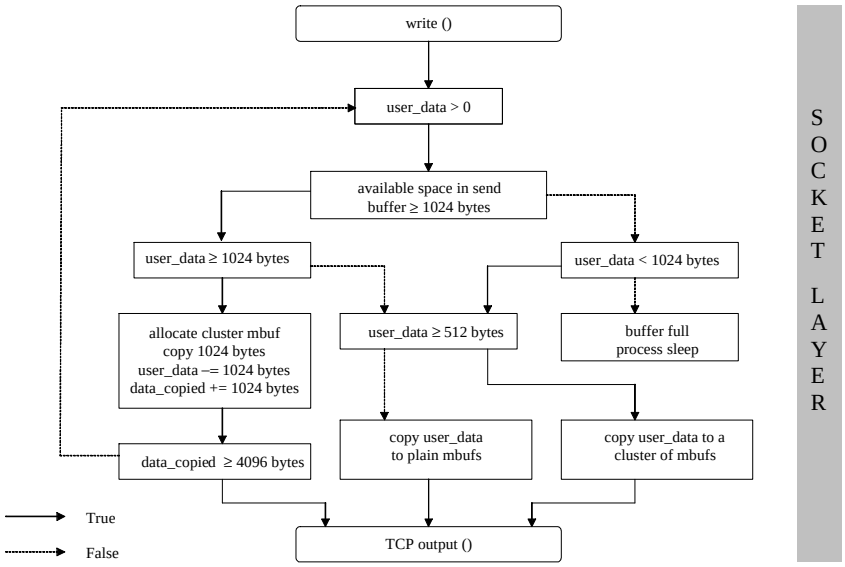

Figure 15.1. Socket layer data copy rules in BSD.

15.2.2 Nagle algorithm and delayed acknowledgment algorithm

Figure 15.2 shows the Nagle algorithm implemented in SunOS 4.x. When there is no outstanding unacknowledged data or the Nagle algorithm is off, the sender can send a segment of any size as long as it is permitted by the usable window size. Otherwise, the unsent data needs to wait in the send buffer if the data unsent is less than both MSS and half of the maximum usable window size. In this case, the sender waits either for more data delivered from the socket layer or for acknowledgments.

The delayed acknowledgment strategy, on the other hand, delays sending acknowledgments until they can be piggybacked onto either a data segment or a window update packet. For example, in the SunOS implementation, a separate window update with a piggybacked acknowledgment will be sent if the window

can slide more than either (a) 35% of the receive buffer size or (b) two MSSes of the size. Even if both conditions are not met, a delayed acknowledgment timer allows sending 1 acknowledgment every 200 ms.

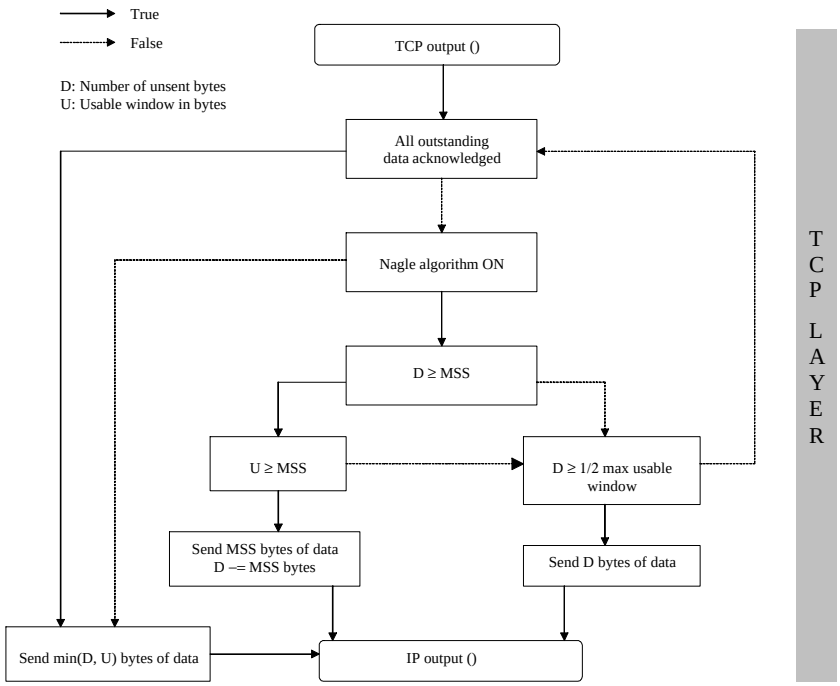

Figure 15.2. The Nagle algorithm.

15.2.3 An Experimental Setup

Unlike the ATM network setup used in [14], in this section we demonstrate that the throughput experiments can be performed in a single workstation (running SunOS 4.1.3 in our case). To allow communication between a sending process and a receiving process, both running in the same machine, we make use of the loopback interface as a logical data link between them. Therefore, the kernel performs complete data processing in SunOS 4.1.3's transport and network layers, and the loopback driver redirects packets sent to the receiving process back to an appropriate input queue. It turns out that changing the loopback driver's MTU is not a trivial task. Our solution is to perform a "software hijack" by putting a loopback MTU adjustment request (a few lines of code) in *tcp_trace.c*, a debug function that can be initiated via *SO_DEBUG* socket option at the socket level. We also wrote a socket program *lperf* to measure SunOS 4.1.3 memory-to-memory TCP through-

put. The program uses BSD socket interface to perform inter-process communication. Upon execution, the program forks a receiver process to collect all sender data via the loopback routine. The sender transmits a continuous data stream, and the established TCP connection would not terminate until the sender finishes sending data.

S \ *R*	4 KB	8 KB	16 KB	24 KB	32 KB	40 KB	48 KB	52 KB
4 KB	23.49	31.61	0.16	0.16	0.16	0.161	0.16	0.16
8 KB	28.33	30.28	0.16	0.16	0.16	0.16	0.16	0.16
16 KB	27.89	34.97	36.24	0.49	0.33	0.47	0.47	0.47
24 KB	27.19	35.42	34.81	41.15	40.83	40.50	0.75	0.75
32 KB	27.63	34.63	35.41	39.60	39.58	38.92	38.46	38.50
40 KB	26.58	34.63	34.97	38.81	39.73	39.39	39.57	38.73
48 KB	27.35	34.50	34.55	38.21	39.61	39.31	39.47	39.09
52 KB	27.01	34.23	34.07	38.35	39.29	39.23	38.89	40.27

Class I deadlocks
Class II deadlocks
Class III deadlocks

Table 15.1. Throughput measurements for a TCP connection with an MSS of 9148 bytes (in Mbps).

The throughput measurements are based on two timestamps generated from the *gettimeofday* system call. The start time is taken at the instant when the sender makes the *write* system call, and the end time is taken at the instant when the sender completes the data transmission. The TCP throughput is computed by dividing the total amount of application data sent by the difference between the two timestamps. In Table 15.1, we present the throughput measurements with the MSS set to 9148 bytes. Each data is an average value computed from 10 independent experiments. All measurements assume zero connection setup time, no packet losses, and a bulk data transfer. Moreover, there is a delay of 180 seconds between experiments in order to alleviate the CPU loading. As shown in the table, throughput deadlocks occur in the shaded region, which are contributed from three different sources, as will be explained next.

15.2.4 Three Classes of Throughput Deadlocks

As explained in Section 15.1, all throughput deadlocks are resulted from a circular-wait condition between a TCP sender and a TCP receiver. Moldeklev and Gunningberg classified the causes for the throughput deadlock into (I) deadlocks predictable from the acknowledgment strategy, (II) deadlocks caused by the socket copy rule and the Nagle algorithm, and (III) deadlocks caused by the timer acknowledgment and the Nagle algorithm [3]. To summarize, class I deadlock occurs if Equation (1) is satisfied, and classes II and III deadlocks occur if Equation (2) is satisfied.

$$S < \min\{2\ \text{MSS}, 0.35\ R\}. \quad (1)$$

$$S < \min\{3\ \text{MSS}, 0.35\ R + \text{MSS}\}, \quad (2)$$

where S and R are the send socket buffer size and receive socket buffer size, respectively.

Equation (1) represents a sufficient deadlock condition for which a maximum sized segment sent by the sender cannot trigger acknowledgments from the receiver. Thus, this type of deadlock depends only on the socket buffer size combination, but not on other implementation issues, such as data copying rules.

Even when Equation (1) does not hold, class II and III deadlocks could still occur according to Equation (2), and these two classes, unlike the first one, depend also on other implementation details. Moreover, the exact causes for classes II and III deadlocks are subtly different. In class II deadlocks, a sender may immediately push out a nonMSS-sized segment of size d because of the data copying rule, thus leaving $S - d$ for buffering new data in the send buffer. As a result, sufficient conditions for deadlocks are given by (i) $S - d <$ MSS and (ii) $d < \min\{2\ \text{MSS}, 0.35\ R\}$. The Nagle algorithm and condition (i) prevent the sender from sending more segments. The delayed acknowledgment strategy and condition (ii), on the other hand, prevent the receiver from acknowledging immediately. S/R = 8 KB/16 KB and 16 KB/40 KB are examples of this class of deadlocks (see Table 15.1).

Class II deadlocks could not occur in S/R = 16 KB/24 KB, 16 KB/32 KB, 24 KB/48 KB, and 24 KB/52 KB, because the sender has enough buffer space to compose an MSS-sized segment. However, class III deadlocks could still occur to them when a sender receives a timer-triggered acknowledgment from the receiver. To be specific, let d be the amount of outstanding segments sent by the sender at a certain time. After that, the sender receives a timer-triggered acknowledgment, which generally could acknowledge any amount of the outstanding segments, and let $a \leq d$ be the amount of segments acknowledged by the timer-triggered acknowledgment. Similar to class II deadlocks, sufficient conditions for deadlocks are given by (i) $S - (d - a) <$ MSS, (ii) $d - a < \min\{2\ \text{MSS}, 0.35\ R\}$. These two conditions are again reduced to Equation (2).

Based on Equation (2), a straightforward solution to avoiding throughput deadlocks is to choose an S/R that violates Equation (2). That is, a static approach is to set the send socket buffer to 3 MSS, regardless of the receive socket buffer size. For 10/100-Mbps Ethernet, for example, only 4380 bytes of socket memory

to meet the requirement. However, this requirement becomes more demanding in the ATM networks with MSS equal to 9148 bytes, where a sender needs to reserve more than 27 KB of buffer memory for a single TCP connection. When the sender maintains a large number of concurrent connections, the memory allocated to the TCP communication would be too enormous to support. Further, this solution is still dependent on the network speed, and the buffer requirement becomes more stringent as the Internet's speed continues to increase.

Another method of violating Equation (2) is to let the sender obtain receive socket buffer information, but this requires additional support from the TCP protocol. Yet another obvious solution is to turn off the Nagle algorithm on the sender side and to explicitly acknowledge every segment received, i.e., without delayed acknowledgment. This solution will clearly reintroduce the small-segment problems that the Nagle algorithm and the delayed acknowledgment strategy were originally designed to solve. To sum up, the inadequacies of these solutions motivate us to examine better solutions to eliminate the deadlock problems, to be discussed next.

15.3 AN ADAPTIVE ACKNOWLEDGMENT ALGORITHM (A^3)

We have identified three important requirements for any solution to the deadlock problem. First, the solution must cater to heterogeneous receivers, which may or may not implement the new solution. Second, an implementation of the solution must involve only a minimal change in the coding. Third, the solution must be adaptive to the network size and the amount of socket buffers available. All three requirements above ensure that the solution is compatible with the current TCP and is immediately deployable. Moreover, the requirements imply that the solution is scalable to the rapid development of the Internet in the future. Based on these requirements, we expect that the new algorithm is a receiver-side SWS avoidance algorithm and the sender-side SWS avoidance algorithm should remain unchanged. In other words, the receiver is solely responsible for ensuring deadlock-free operations, and it only assumes that the sender transmission behavior is governed by a window-based flow control mechanism.

Based on the design requirements, we propose a new A^3 that determines when to send acknowledgments based on the sender's behavior. A key quantity maintained by an A^3-receiver is referred to as *maximum unacknowledged data size* (MUDS) in terms of bytes, which is defined to be the maximum amount of data continuously sent by the sender when the receiver is not acknowledging. When deadlocks do not occur and there is no network congestion, the MUDS is upper bounded by $\min\{S, R\}$. In general, the MUDS' exact value depends on the buffer sizes, data copying rule, and other implementation issues. For example, the MUDS is given by MSS + 4 KB for MSS + 4 KB $\leq S <$ $\min\{3 \text{ MSS}, 0.35 R + \text{MSS}\}$ (class II deadlocks). Given an MUDS estimate, an A^3-receiver sends an acknowledgment whenever the segments received reach or exceed 35% of the estimated

MUDS. Thus, an A^3-receiver no longer relies on the receiver's actual buffer size and the actual value of the MSS, and it is able to adapt to various network MTUs and different combinations of send and receive buffer sizes. Moreover, it is clear that the new acknowledgment algorithm removes all three classes of deadlocks by having the receiver promptly send back acknowledgments.

One approach to estimating the MUDS is to perform "samplings" periodically by an A^3-receiver. During a sampling, the receiver keeps track of the amount of segments received, and it will not acknowledge any segments. Based on the sampling information, an A^3-receiver estimates the sender's MUDS. Thus, an A^3-receiver alternates between sampling periods and nonsampling periods.

The packet samplings incur overheads and delay, because acknowledgments will not be sent during these periods. Thus, controlling the length of the sampling periods and the frequency of performing samplings are major factors influencing the A^3's performance. For this purpose, an A^3-receiver is equipped with the following timer, counter, and thresholds. We also show in Figure 15.3 an A^3-receiver's state transition diagram during sampling periods.

- Sampling period timer (SP_timer): A timer to control the length of a sampling period
- Sampling period threshold (SP_t): A value (in time units) used in conjunction with the SP_timer to control the length of a sampling period
- Inter-sampling period counter (ISP_counter): A counter to control the inter-sampling period
- Inter-sampling period threshold (ISP_t): A value (in number of packets) used in conjunction with the ISP_counter to control the inter-sampling period

15.3.1 Inter-Sampling Period Control and Sampling Initialization

The value of ISP_t determines the time interval between two consecutive samplings and it is set according to the following algorithm, where $MUDS_{current}$ and $MUDS_{previous}$ are the current and previous estimates of the MUDS, respectively.

if ($MUDS_{previous}$ does not exist)
    ISP_t = 10;
else {
    if (|$MUDS_{current}$ - $MUDS_{previous}$| < MSS) {
        ISP_t = 3 ISP_t;
        the acknowledgment threshold remains unchanged;
    } else {
        ISP_t = 10;
        if ($MUDS_{current}$ > $MUDS_{previous}$)
            the acknowledgment threshold = 0.35 $MUDS_{current}$;
    }
}

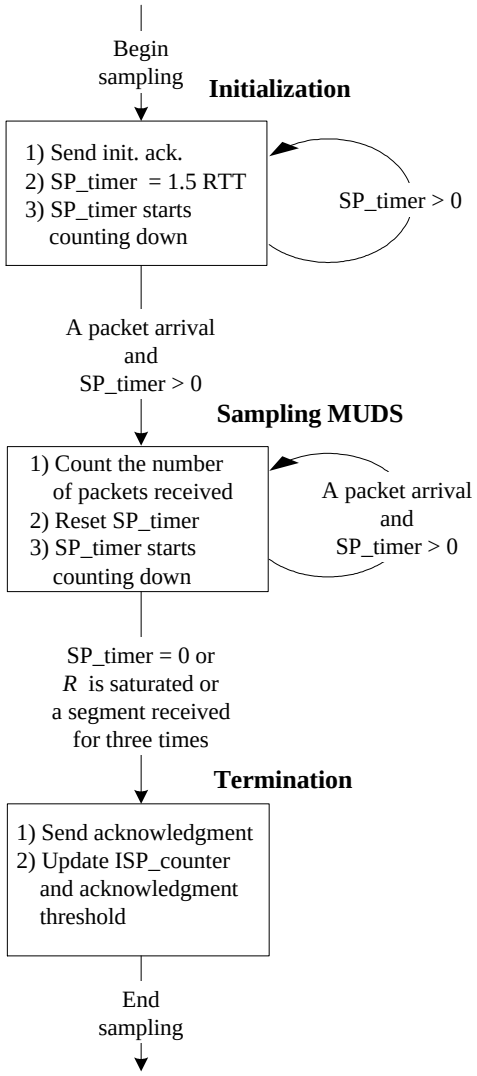

Figure 15.3. An A^3-receiver's state diagram during a sampling period.

The ISP_t is set to a constant (10 in the algorithm) as soon as a TCP connection is established, and the ISP_counter is initialized to ISP_t and begins to count down, i.e., the counter is decremented whenever a packet is received. A packet

sampling starts when the ISP_counter's value drops to zero. At the end of a packet sampling, the counter may be reset to ISP_t again. However, the counter is tripled if $MUDS_{current}$ is "close" to $MUDS_{previous}$ because this is a clear indication that $MUDS_{current}$ is accurate and the number of samplings should be decreased.

At the beginning of a packet sampling, the receiver will first send a maximum window update with an acknowledgment for all the segments received in the buffer. The acknowledgment and window update therefore allow the sender to transfer the maximum amount of data permitted by the sender-side SWS avoidance mechanism. At the same time, the SP_timer is initialized to 1.5 RTT (round-trip time) and the timer starts counting down. This initialization allows adequate time to receive the packets in transit and the packets sent upon receiving the receiver's first acknowledgment. Once a sampling is started, the receiver is prohibited from further acknowledging the incoming data until the sampling period ends.

15.3.2 Sampling Period Control

A TCP sender may dispatch an arbitrary amount of segments bounded by the sender-side flow control mechanism and the send buffer size. The receiver carefully records the amount of data received, and discards retransmissions. The sampling procedure terminates when (1) the receive buffer is saturated (or close to saturation) or (2) a single segment is received three times in a row (this is a strong indication of timeout or packet lost). Otherwise, the sampling will continue until (3) the connection has been inactive for a certain period of time, i.e., when the SP_timer becomes zero.

We have mentioned that the SP_timer is initialized to 1.5 RTT when a connection is established. The SP_timer is reset to the threshold SP_t whenever a packet is received. The timer then starts counting down and the receiver terminates the sampling period when the timer reaches zero. The choice of the threshold SP_t is important. Having a large threshold value will result in a performance penalty since no acknowledgments are sent during a sampling period. On the other hand, having a small value may result in terminating the sampling prematurely and, as a result, the receiver underestimates the MUDS. Here we use inter-packet arrival time (IPAT) to determine the threshold by setting the SP_t to 1.3 IPAT. In other words, the receiver concludes that the sender has sent a maximum amount of segment permitted by the sender-side SWS avoidance mechanism if there are no other packet arrivals after a period of 1.3 IPAT. To do so, the receiver continuously measures the IPAT for the incoming packets. For every five packets received, the receiver discards the maximum and minimum IPAT samples and takes the average of the remaining three values as the current IPAT estimate. The IPAT clearly depends on the transmission delay, sender packetization delay, and other factors. By computing the IPAT for every five packets, we are hoping to obtain the most updated estimate for the packet interarrival pattern.

One exception to resetting the SP_timer to 1.3 IPAT upon receiving a packet is when the remaining time in the timer upon a packet arrival is larger than 1.3

IPAT. This will occur if a transit packet arrives at the receiver shortly after the initialization of a packet sampling. If the SP_timer is immediately reset to 1.3 IPAT in this case, the packet sampling may terminate prematurely since the first acknowledgment sent by the receiver at the initialization stage may not arrive at the sender in time. Therefore, the A^3 ensures that the sampling period is at least 1.5 RTT.

The following summarizes the algorithm of updating the SP_timer when a packet is received.

if (SP_timer > 1.3 IPAT upon a packet arrival)
    SP_timer continues to count down;
else
    SP_timer = 1.3 IPAT and the timer starts counting down.

15.3.3 Computing the Thresholds

During and at the end of a sampling period, an A^3-receiver is required to update the ISP_t, SP_t, and the acknowledgment threshold. The updates for the ISP_t and SP_t have already been discussed in the previous two sections. As for the acknowledgment threshold, a sender's actual MUDS normally does not change significantly in the congestion avoidance phase. Thus, we only update the MUDS estimate when it is increased. To follow the delayed acknowledgment algorithm in the SunOS, we set the acknowledgment threshold to 35% of the MUDS estimate, as shown in the algorithm in Section 15.3.1.

15.3.4 Performance Evaluation

To compare the performance of an A^3-receiver with that of a typical TCP receiver equipped with the delayed acknowledgment algorithm, we simulate a TCP connection with an MSS of 9148 bytes. Each endpoint in the simulator is a 4.3 BSD Unix terminal, which observes the essential 4.3 BSD socket mechanism, including the data copying rules and TCP segment buffering. Our simulated TCP channel takes no time to establish a connection, and the packet loss is assumed negligible. The TCP performance is measured through a *ftp* session from a sender to a receiver. The throughput is computed by the total amount of data transferred by the total transmission time. The experiments cover 8×8 send-receive buffer size combinations ranging from 4 KB to 52 KB, and each individual experiment is a continuous (back to back) transmission of 10-MB data. Every throughput measurement reported in Table 15.2 is an average of 25 independent runs.

Our simulation results show that the A^3 has successfully reclaimed the TCP performance in the original deadlock regions. Additionally, the adaptive acknowledgment threshold improves the performance in other nondeadlock areas. This is due to an earlier response from the receiver when the A^3 is used, which results in a more efficient pipelining of the data from the sender. On the other hand, there is a marginal performance penalty caused by the A^3's operational overheads for small receivers, e.g., R = 4, 8 KB. Consider S/R = 16 KB/4 KB, the receiver acknowl-

edgment threshold is 1.4 KB, and the sender maximum usable window size is 4 KB. This suggests that a single data segment (4 KB) from the sender is enough to trigger a receiver acknowledgment from either acknowledgment algorithm. Unfortunately, an A^3-receiver requires more time to compute the IPAT, sampling threshold, and acknowledgment threshold, and these additional overheads introduce performance degradation, as shown in the lower left triangular regions in Table 15.2, where the send buffer is relatively large when compared with the receive buffer.

R / *S*	4 KB (1)	4 KB (2)	4 KB (3)	8 KB (1)	8 KB (2)	8 KB (3)	16 KB (1)	16 KB (2)	16 KB (3)	24 KB (1)	24 KB (2)	24 KB (3)
4 KB	20	20	21	20	20	21	6	20	20	6	20	20
8 KB	21	20	20	25	20	20	6	25	25	6	25	26
16 KB	21	20	20	41	41	41	55	50	50	19	50	50
24 KB	21	20	20	41	40	40	79	80	80	74	87	89
32 KB	20	20	21	41	40	40	81	80	82	80	89	95
40 KB	20	20	21	41	40	41	81	80	80	99	98	103
48 KB	20	20	20	41	40	42	81	80	81	99	99	102
52 KB	21	20	21	41	40	41	82	80	80	100	98	98

R / *S*	32 KB (1)	32 KB (2)	32 KB (3)	40 KB (1)	40 KB (2)	40 KB (3)	48 KB (1)	48 KB (2)	48 KB (3)	52 KB (1)	52 KB (2)	52 KB (3)
4 KB	6	20	20	6	20	21	6	20	20	6	20	21
8 KB	6	25	25	6	25	26	6	25	25	6	25	26
16 KB	19	50	50	19	50	50	19	50	50	19	50	50
24 KB	73	87	88	74	87	87	26	87	88	26	87	87
32 KB	89	94	106	89	98	107	94	102	115	94	99	118
40 KB	99	99	103	141	134	137	111	139	144	111	137	146
48 KB	100	104	102	151	150	159	160	161	170	143	161	168
52 KB	100	99	105	152	152	160	162	168	165	171	162	170

(1) Delayed acknowledgment algorithm
(2) The A^3 according to Section 15.3
(3) The A^3 with a fewer number of samplings
Shaded region: Deadlock region for the delayed acknowledgment algorithm

Table 15.2. Throughputs of a TCP connection with an MSS of 9148 bytes (in KB per simulation second).

A possible way to further improving the A^3's throughput performance is therefore to reduce the number of samplings required for computing the MUDS.

There are two ways to achieve that: (1) increase the inter-sampling period and (2) decrease each sampling period. In Table 15.2, we also present simulation results for a modified A^3 in which ISP_t = 3 ISP_t is changed to ISP_t = 4 ISP_t under the if-statement, and ISP_t = 10 is changed to ISP_t = 0.5 ISP_t under the else statement in the algorithm presented in Section 15.3.1. Although the number of samplings decreases significantly (not shown in the table), the throughput improvement is not significant. In the original A^3, the overhead incurred from additional samplings is compensated by a gain in throughput. In the modified A^3, a loss of gain in throughput by not sampling as frequently is compensated by reducing the sampling overhead.

15.4 EFFECT OF NETWORK CONGESTION ON THE A^3

The A^3 was designed without network congestion in mind. In this section, we show and explain why the A^3 fails to ensure throughput deadlock-free when a TCP connection experiences network congestion. Then we propose an enhancement to the A^3, referred to as congestion-sensitive A^3 (CSA^3), to ensure deadlock-free even in the presence of network congestion.

It is well known that a TCP sender detects possible network congestion either by receiving three duplicate acknowledgments (fast retransmission) or by retransmission timeouts. In either case, the sender retransmits the oldest segment that has not been acknowledged. At the same time, the sender also adjusts its congestion window size. In the former case, the sender exercises fast recovery in which the congestion window is reduced to only half of the current value. In the latter, the window size is reset to one MSS. Both cases effectively reduce the sender's sending rate, and consequently the A^3-receiver's MUDS estimate overestimates the actual value. Therefore, the receiver may not be able to receive enough data to trigger acknowledgment, and throughput deadlock recurs.

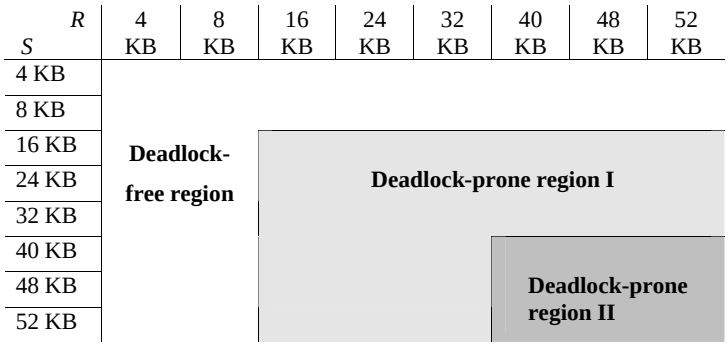

Table 15.3. A classification of send-receive buffer combinations for an MSS of 9148 bytes for A^3-receivers.

15.4.1 A Classification of Operating Regions

Before introducing the CSA^3, it is convenient to divide the 64 send-receive buffer combination into three regions assuming that the MSS is 9148 bytes, as shown in Table 15.3. In the deadlock-free region, throughput deadlocks will not occur to TCP connections equipped with A^3-receivers in the presence of network congestion and we postpone the explanation to the next section. On the other hand, throughput deadlocks recur in the two deadlock-prone regions when there is network congestion. The throughput degradation is more serious in the deadlock-prone region II than that in the deadlock-prone region I.

A. Deadlock-Free Region

Figure 15.4 shows the throughput for a scenario in the deadlock-free region (S/R = 4 KB/4 KB). Network congestions of short durations are injected into the simulation at several time instants. The two curves, with and without congestion, are overlapped before the network congestion first occurred at around 500 seconds, and when the time is more than 2000 seconds. Between the two times, very small throughput drops are noted in the curve indicated by "A^3 under congestion" because the dropped segments need to be retransmitted. Other scenarios in this region, though not shown here, actually exhibit very similar behavior as Figure 15.4. Specific points about this set of experiments are as follows.

When the send buffers are small (4 KB and 8 KB), the sender's MUDS obviously cannot be very large. In fact, owing to the data copying rule, the sender always pushes out a 4-KB segment, and the next segment, if any, can be sent out only after receiving an acknowledgment from the receiver. As a result, the network congestion only causes the sender to time out and to retransmit the dropped segments (therefore the drops in the throughput), but it will not affect the MUDS value. Therefore, the receiver can promptly send back acknowledgments for any segments received, and no deadlocks occur.

When the send buffers are large enough (≥ 16 KB), the send buffer allows the sender to send more than one segment at a time. However, since the receive buffers are small in this region (4 KB and 8 KB), the sender's usable window is limited by the receive buffer size, which is advertised to the sender in the TCP segments. Thus, the sender's MUDS is also given by 4 KB. Similar to the small send-buffer cases, network congestion will only delay the recipient of the dropped segments, but will not affect the MUDS estimates. As a result, no deadlocks occur.

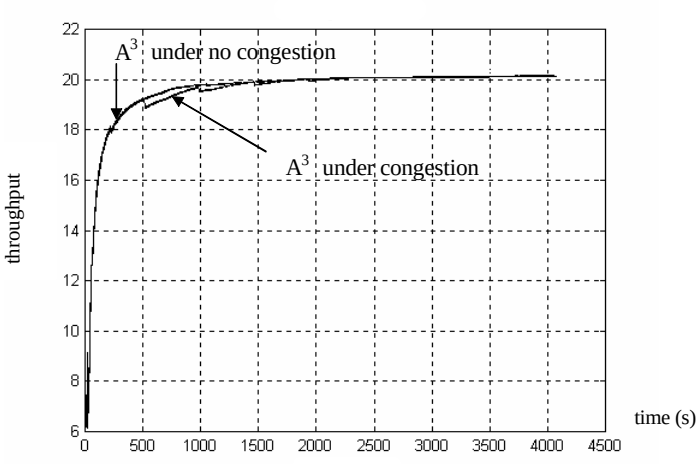

Figure 15.4. Throughput of a 4-KB sender with a 4-KB A^3-receiver.

B. Deadlock-Prone Region II

Figure 15.5 shows a scenario belonging to the deadlock-prone region II. The upper curve corresponds to the no-congestion case. Therefore, the throughput increases steadily as the window size continues to increase. However, congestions are injected into the simulation for the lower curve at three different times. Whenever congestion occurs, the throughput drops significantly, because the A^3-receiver has over-estimated the MUDS. When the sender times out due to congestion, the sender resets its congestion window size to one MSS. Therefore, without knowing the time-out event, the A^3-receiver continues to use the MUDS estimate, which is obtained before congestion, for computing the acknowledgment threshold. As a result, a deadlock recurs and the A^3-receiver acknowledges segments upon timing out its delayed acknowledgment timer. After updating the MUDS based on the new samplings, the A^3-receiver is again able to avoid deadlocks, and the throughput subsequently increases.

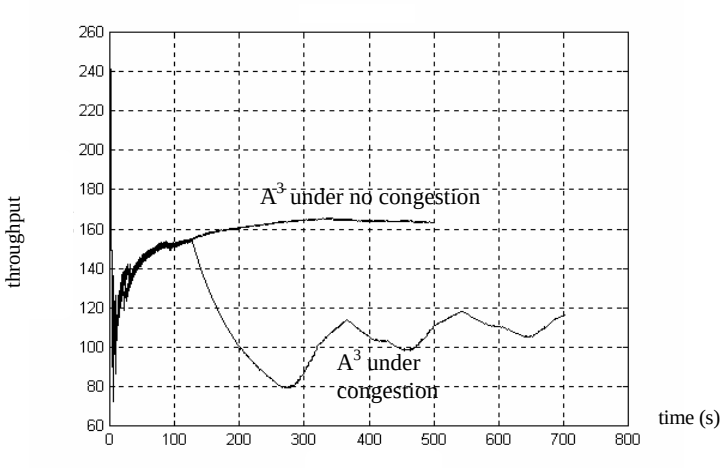

Figure 15.5. Throughput of a 52-KB sender with a 52-KB A^3-receiver.

C. Deadlock-Prone Region I

Figure 15.6 shows a scenario belonging to the deadlock-prone region I. Similar to the region II, throughput drops are observed in both scenarios. However, the magnitudes of the throughput degradation are not as significant and the throughput can also be recovered relatively quickly. Although the buffer sizes are much smaller than those in region II, the over-estimations of the MUDS are significant enough that deadlocks recur in this region.

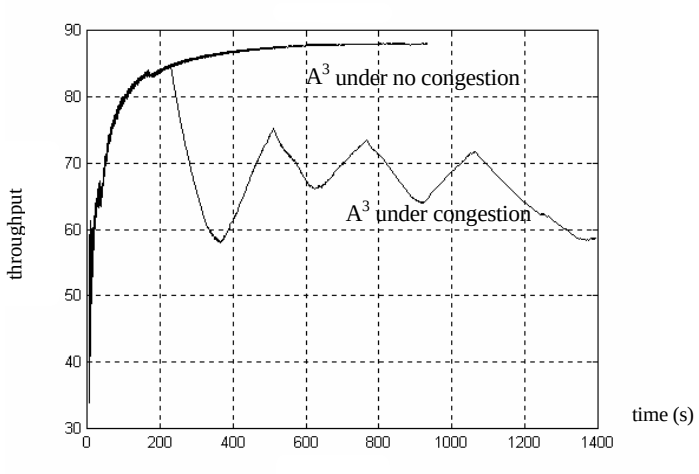

Figure 15.6. Throughput of a 24-KB sender with a 24-KB A^3-receiver.

15.4.2 Congestion-Sensitive A^3 (CSA^3)

A straightforward approach to resolving the congestion problem is to reset the MUDS and to perform samplings all over again to find the new value whenever congestion is detected. However, the main disadvantage of this approach is that deadlock still exists before the sampling is completed. As a result, the sampling needs to run for a long time, and therefore the overhead incurred would be very significant. Because of that, our approach proposed here is to enhance the A^3 so that the enhanced A^3 is able to react to network congestion.

A. A Slow-Start-Like Algorithm for Updating the MUDS

A CSA^3-receiver is now required to infer network congestion based on the information received, and to reduce its MUDS estimate based on a procedure very similar to TCP senders' slow-start and congestion avoidance algorithms. Specifically, whenever a CSA^3-receiver suspects network congestion based on the information to be discussed in the latter part of this section, it remembers MUDS/2 in a variable *r_ssthresh* and resets MUDS to one MSS. One exception to this is when the current MUDS is smaller than MSS, and this is possible when either the send buffer or receive buffer is smaller than MSS, such as those cases in the deadlock-free region. In other words, this MUDS updating procedure does not apply to the scenarios in the deadlock-free region.

After resetting MUDS to one MSS, the CSA^3-receiver doubles MUDS for every nonduplicate acknowledgment sent out. This phase corresponds to the sender's slow-start phase. When the MUDS finally reaches or exceeds *r_ssthresh*, it is further increased linearly, which is similar to senders' congestion avoidance phase. When the MUDS is finally increased to the old value before congestion, the CSA^3-receiver continues to update the MUDS in the same way as for an A^3-receiver.

The MUDS updating algorithm upon detecting congestion is summarized below.

```
r_ssthresh = MUDS / 2;
while (a new acknowledgment sent out by the receiver) {
    if (MUDS £ r_ssthresh)
        MUDS = MUDS * 2;
    else {
        MUDS += MSS * (MSS / MUDS);
        if (MUDS ³ r_ssthresh * 2) exit;
    }
}
```

There are altogether three conditions that would trigger a CSA^3-receiver to update its MUDS estimate according to the algorithm above:

1. When a router experiences short-term congestion, it drops a small number of segments. As a result, it is very likely that a CSA^3-receiver is able to receive three duplicate acknowledgments, and it will perform fast retransmit and fast

recovery. Moreover, it will update its MUDS estimate according to the slow-start-like algorithm.

2. When a router experiences serious congestion, it drops a large number of segments. As a result, a CSA^3-receiver may not be able to receive enough duplicate acknowledgments to perform fast retransmit and fast recovery. Therefore, the sender will finally time out and retransmit lost segments. This also suggests that a CSA^3-receiver needs to time out its MUDS estimate after a certain inactivity period. Upon time out, it performs the same slow-start-like algorithm to update the MUDS.
3. Besides dropping segments, congested routers may also drop acknowledgments on the reverse path. As a result, the sender will eventually time out and retransmit lost segments, and a CSA^3-receiver may receive duplicate data segments. Therefore, the CSA^3-receiver also performs the slow-start-like algorithm when it receives duplicate segments.

B. Performance Evaluation of the CSA^3

In Figures 15.7-15.9, we present simulation results to compare the throughput performance of the A^3 and CSA^3 in the presence of network congestion. Since the A^3's throughput is minimally affected by congestion in the deadlock-free region, as shown in Figure 15.4, we only consider the two deadlock-prone regions in this section. The simulation settings are the same as those in Section 15.4.1. Network congestions occur at several time instants, and the TCP sender retransmits lost segments upon timeout.

Among all three scenarios (one in region I and two in region II), the improvement in throughput is most significant in the deadlock-prone region II when the CSA^3 is employed. Since both send and receive buffers are large in this region, the MUDS value can be quite high when there is no congestion. Therefore, the CSA^3's receiver also keeps a large MUDS estimate. When network congestion occurs, the sender resets the congestion window to one MSS. Both the CSA^3-receiver and A^3-receiver are unable to promptly acknowledge new segments due to the over-estimation of the sender's MUDS, thus resulting in throughput drops in both cases. However, the CSA^3-receiver is able to recover from throughput drop much faster than the A^3-receiver, because the former promptly reduces the MUDS estimate and therefore can send a new acknowledgment much earlier than the A^3-receiver can. In TCP, the sender's congestion window size increases based on a self-clocking acknowledgment mechanism. The faster new acknowledgments are received, the faster the window is opened up. As a result, the CSA^3-receiver can recover from the throughput loss quickly, as shown in Figure 15.7.

The throughput behaviors shown in Figure 15.8 and 15.9 for region I, on the other hand, are quite similar. The CSA^3 attains similar throughput as the A^3 at the beginning, but the former again responds to and recovers from network congestion much faster than the A^3. In this region, the MUDS estimate may or may not be large enough to cause deadlocks. Apparently, when the first congestion occurs, the CSA^3 receiver is unable to detect the network congestion; therefore, there is no

throughput gain by using the CSA^3. However, in the latter congestion the CSA^3 receiver is able to detect congestion and to respond to it promptly. This shows that the CSA^3 does not impose any penalty on the throughput performance even though it occasionally fails to detect congestion.

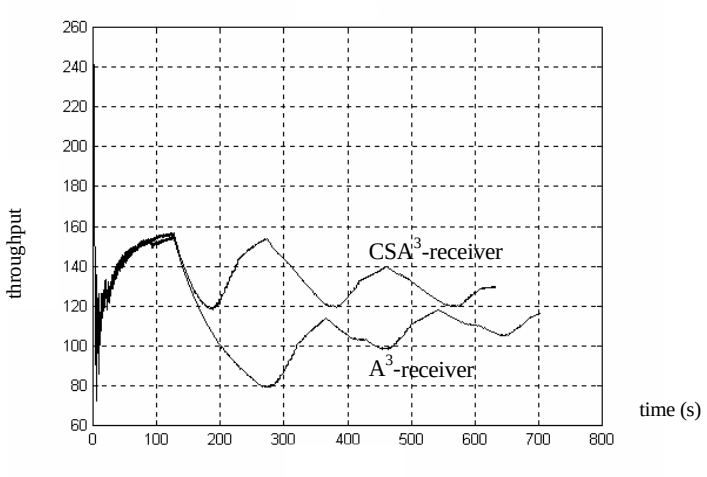

Figure 15.7. Throughput of a 52-KB sender with a 52-KB A^3/CSA^3-receiver under congestion.

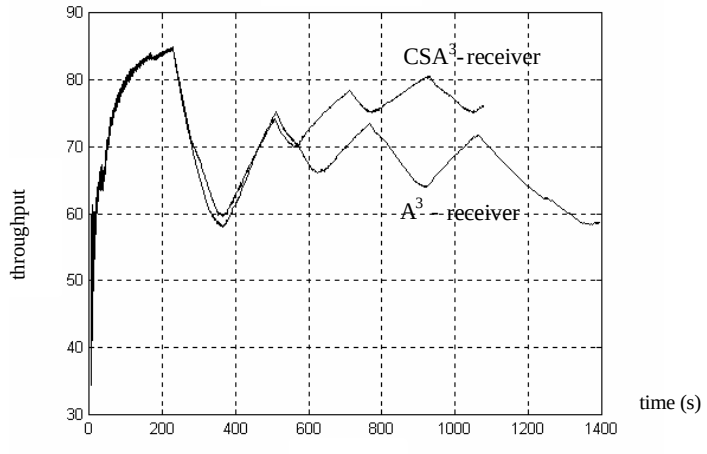

Figure 15.8. Throughput of a 24-KB sender with a 24-KB A^3/CSA^3-receiver under congestion.

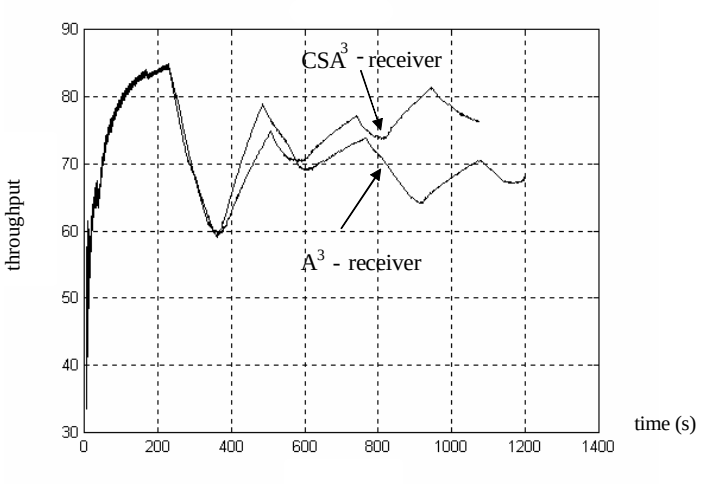

Figure 15.9. Throughput of a 24-KB sender with a 48-KB A^3/CSA^3-receiver under congestion.

15.5 CONCLUSIONS AND FUTURE WORK

In this paper we have presented the TCP throughput deadlock problems in high-speed networks. This problem becomes very serious as the end-to-end network speed continues to increase. We have proposed the A^3 and CSA^3 to overcome this problem. We have shown by simulations that they have successfully reclaimed all deadlock regions for noncongested networks and congested networks, respectively. Currently, we are implementing them in Linux systems and we will put them in field tests. Another important area to study concerns fairness between the CSA^3 receivers and nonCSA^3 receivers. It remains to be seen whether the new acknowledgment algorithms impose any unfairness toward those that do not implement them.

Acknowledgment

This work is partially supported by The Hong Kong Polytechnic University Research Grant G-S909.

REFERENCES

[1] N. Ghani and S. Dixit, "TCP/IP Enhancements for Satellite Networks," *IEEE Commun. Mag.*, pp. 64-72, July 1999.
[2] C. Charalambous, V. Frost, and J. Evans, "Performance Evaluation of TCP Extensions on ATM Over High Bandwidth Delay Product Networks," *IEEE Commun. Mag.*, pp. 57-63, July 1999.
[3] C. Partridge and T. Shepard, "TCP/IP Performance Over Satellite Links," *IEEE Network Mag.*, pp. 44-49, Sept/Oct. 1997.
[4] H. Balakrishnan and V. Padmanabhan, "How Network Asymmetry Affects TCP," *IEEE Commun. Mag.*, vol. 39, no. 4, pp. 60-67, Apr. 2001.
[5] G. Xylomenos, G. Polyzos, P. Mahonen, and M. Saaranen, "TCP Performance Issues Over Wireless Links," *IEEE Commun. Mag.*, vol. 39, no. 4, pp. 52-59, Apr. 2001.
[6] C. Papadopoulos and G. Parulkar, "Experimental Evaluation of SUNOS IPC and TCP/IP Protocol Implementation," *IEEE/ACM Trans. Networking*, pp. 199-216, April, 1993.
[7] V. Paxson, "Automated Packet Trace Analysis of TCP Implementations," *Proc. ACM SIGCOMM*, pp. 167-179, 1997.
[8] J. Semke, J. Mahdavi, and M. Mathis, "Automatic TCP Buffer Tuning," *Proc. ACM SIGCOMM*, pp. 315-323, 1998.
[9] D. Clark, "Window and Acknowledgment Strategy in TCP," *RFC 813*, July 1982.
[10] J. Nagle, "Congestion Control on TCP/IP Internetworks," *RFC 896*, Jan. 1984.
[11] R. Braden, "Requirements for Internet Hosts—Communication Layers," *RFC 1122*, Oct. 1989.
[12] J. Heidemann, "Performance Interactions Between P-HTTP and TCP Implementations," *ACM Computer Communication Review*, vol. 27, no. 2, pp. 65-73, Apr. 1997.
[13] J. Mogul and G. Minshall, "Rethinking the TCP Nagle Algorithm," *ACM Computer Communication Review*, vol. 31, no. 1, pp. 6-20, Jan. 2001.
[14] K. Moldeklev and P. Gunningberg, "How a Large ATM MTU Causes Deadlocks in TCP Data Transfers," *IEEE/ACM Trans. Networking*, vol. 3., no. 4., pp. 409-422, Aug. 1995.
[15] D. Comer and J. Lin, "TCP Buffering and Performance over an ATM Network," *Internetworking: Research and Experience*, vol. 6, pp. 1-13, May 1995.
[16] Y. Murayama and S. Yamaguchi, "A Proposal for a Solution of the TCP Short-Term Deadlock Problem," *Proc. 12th Intl. Conf. Information Networking*, pp. 269-274, 1998.
[17] G. Wright and W. Stevens, *TCP/IP Illustrated*, vol. 2, Addison-Wesley, 1995.