On the Generation and Use of TCP Acknowledgments

Mark Allman
NASA Lewis Research Center/Sterling Software
mallman@lerc.nasa.gov

Abstract

This paper presents a simulation study of various TCP acknowledgment generation and utilization techniques. We investigate the standard version of TCP and the two standard acknowledgment strategies employed by receivers: those that acknowledge each incoming segment and those that implement delayed acknowledgments. We show the delayed acknowledgment mechanism hurts TCP performance, especially during slow start. Next we examine three alternate mechanisms for generating and using acknowledgments designed to mitigate the negative impact of delayed acknowledgments. The first method is to generate delayed ACKs only when the sender is not using the slow start algorithm. The second mechanism, called *byte counting*, allows TCP senders to increase the amount of data being injected into the network based on the amount of data acknowledged rather than on the number of acknowledgments received. The last mechanism is a limited form of byte counting. Each of these mechanisms is evaluated in a simulated network with no competing traffic, as well as a dynamic environment with a varying amount of competing traffic. We study the costs and benefits of the alternate mechanisms when compared to the standard algorithm with delayed ACKs.

1 Introduction

The Transmission Control Protocol (TCP) [Pos81] is the prevalent reliable transport protocol used on the Internet today. TCP data receivers transmit acknowledgments (ACKs) to the data sender as segments arrive. This provides reliability, as the sender retransmits any segments that are not acknowledged by the receiver. Since TCP is a sliding window protocol, incoming ACKs also allow the transmission of new data. Finally, ACKs are used by TCP's congestion control algorithms [JK88] [Ste97] [SAP98] to increase the amount of outstanding data (data that has been sent but not yet acknowledged) the sender is permitted to inject into the network.

As originally outlined in [Pos81], TCP receivers generate an ACK for each incoming segment. These ACKs are *cumulative* and acknowledge all in-order segments that have arrived at the receiver. The redundancy provided by cumulative ACKs protects against ACK loss.

If an out-of-order segment arrives, an ACK is transmitted. However, it will not ACK the incoming segment, but rather a *duplicate ACK* for the last in-order segment that arrived is generated. RFC 1122 [Bra89] defines an optional *delayed acknowledgment* mechanism. Delayed ACKs allow a TCP receiver to refrain from transmitting an ACK for every incoming segment. However, the receiver must send an ACK for every second full-sized segment. In addition, an ACK cannot be delayed for more than 500 ms while waiting for a second full-sized segment to arrive. Since ACKs are cumulative, using delayed ACKs has little impact on transmission reliability. Furthermore, delayed ACKs conserve resources by decreasing the load on the network and the machines that must generate and process these segments.

Paxson [Pax97] found that delayed ACKs are used in a variety of TCP implementations common in the Internet today. Delayed ACKs have been found to have a positive impact on the performance of bulk transfers in certain networks [Joh95]. In addition, [BPK97] shows that using delayed ACKs can increase performance in asymmetric networks. However, delayed ACKs can also reduce performance in certain situations. TCP senders increase the amount of outstanding data injected into the network based on the *number* of ACKs received. Therefore, by reducing the number of ACKs the receiver generates by roughly half, the rate that TCP senders increase the amount of data injected into the network is also reduced.

This paper investigates several alternate ways to generate and utilize TCP acknowledgments that mitigate the negative impact that delayed ACKs can have on performance[1]. The sender employs the *slow*

[1] We do not claim to be the originators of the ACK generation/utilization mechanisms investigated in this paper. These ideas seem to have been independently thought of and discussed by a number of researchers. We were not able to find

start algorithm, which uses incoming ACKs to increase the amount of outstanding data injected into the network at the beginning of a transfer. The first mechanism we investigate is the use of delayed ACKs only when the sender is not using the slow start algorithm. While the sender is using slow start, every segment is acknowledged to provide more ACKs and therefore a more rapid increase to an appropriate window size. Next, we study the use of *byte counting* to increase the amount of outstanding data the sender can inject into the network. This mechanism bases the increase in the amount of outstanding data transmitted on the number of previously unacknowledged bytes acknowledged by each incoming ACK, rather than on the number of incoming ACKs. Our simulations show byte counting to be too aggressive in many situations. This leads to the last mechanism presented, which is a limited form of byte counting.

The remainder of this paper is organized as follows. Section 2 outlines TCP's standard algorithms for increasing the amount of data injected into the network, as well as the alternate mechanisms for generating and using ACKs studied in this paper. Section 3 outlines a set of simple simulations that investigate the differences between the ACK generation and utilization mechanisms in a number of environments. Section 4 investigates the various mechanisms in simulations with competing traffic. Section 5 examines ACK generation and utilization for interactive applications. Finally, section 6 outlines our conclusions and future work on these mechanisms.

2 TCP Congestion Control Algorithms

TCP employs several congestion control algorithms to adjust the amount of data injected into the network based on the amount of network congestion observed. These algorithms provide network stability and prevent congestive collapse [JK88] [FF98]. The standard algorithms used to increase the amount of outstanding data injected into the network are *slow start* and *congestion avoidance* [JK88] [Ste97] [SAP98]. Additionally, TCP Reno includes the *fast retransmit* and *fast recovery* [Jac90] [Ste97] [SAP98] loss recovery algorithms. Our investigation also considers a version of TCP that employs selective acknowledgments (SACKs) [MMFR96]. The SACK version of TCP used in this paper employs fast retransmit and the conservative fast recovery replacement outlined in [FF96].

This section will briefly outline slow start and congestion avoidance and the problems associated with the algorithms in the face of delayed ACKs. In addition, the alternate ACK generation/utilization mechanisms will be outlined.

2.1 Slow Start

The slow start algorithm [JK88] [Ste97] [SAP98] is used to gradually increase the amount of data a TCP sender injects into the network, as well as starting the *ACK clock*. Slow start is used at the beginning of a transfer and following loss detected by TCP's retransmission timer[2]. The TCP sender uses a *congestion window* (*cwnd*) to limit the amount of outstanding data injected into the network to an appropriate level for the current network conditions. The slow start algorithm initializes *cwnd* to 1 segment[3]. This allows the sender to transmit a single data segment. For each ACK received *cwnd* is increased by 1 segment. Slow start is terminated when *cwnd* reaches the receiver's advertised window, or congestion (loss) is detected. Equation 1, where R is the round-trip time (RTT) between the sender and receiver and W_A is the receiver's advertised window (in segments), gives the time the TCP sender spends using the slow start algorithm to increase *cwnd* from 1 segment to the advertised window size when the receiver ACKs each incoming segment [JK88].

$$slow\ start\ time = Rlog_2W_A \tag{1}$$

By comparison, equation 2 gives the approximate time[4] spent by the TCP sender using the slow start algorithm when the receiver implements delayed acknowledgments [PS97].

$$slow\ start\ time \approx 2Rlog_2W_A \tag{2}$$

Note that each of the above equations assume that *cwnd* is able to reach the advertised window before

any formal discussions of the mechanisms in the literature, therefore we credit the research community as a whole for these ideas.

[2]Some TCP implementations also use slow start after a relatively long quiescent period. This use of slow start is not investigated in this paper, as we feel beginning transmission after a long idle period is roughly equivalent to starting transmission at the beginning of a connection. Therefore, we expect the mechanisms outlined in this paper to perform roughly the same in both cases.

[3]In practice, *cwnd* is sometimes stored in terms of bytes. However, for simplicity we will discuss it in terms of segments in this paper.

[4]It is difficult to *exactly* quantify the time required to open *cwnd* in the face of delayed ACKs because of the delayed ACK timer. The timer implementation, the length of the timeout and the RTT between the sender and receiver all interact to make prediction of the exact increase difficult, and beyond the scope of this paper.

detecting congestion and that no ACKs are dropped in the network.

As illustrated by the above equations, using delayed ACKs roughly doubles the time required to increase *cwnd* from 1 segment to the advertised window size. This increase in time required is caused by the reduction in the number of ACKs being sent by the receiver. The slow start algorithm increases *cwnd* by 1 segment for each ACK received. Therefore, reducing the number of ACKs by roughly half increases the time required to open *cwnd* by a factor of 2. The increase in time required by slow start can especially hurt performance for short transfers that complete before the slow start phase is terminated (e.g., short world-wide web pages), as well as transfers over long-delay channels (e.g., satellite links).

Delayed ACKs increase the size of line-rate bursts sent by TCP, while decreasing overall burstiness. Consider the transmission of two segments during slow start. When the receiver ACKs each segment separately, the sender receives 2 ACKs. In response to each ACK, *cwnd* slides by 1 segment and the slow start algorithm increases *cwnd* by 1 segment. This leads to a transmission of 2 back-to-back segments for each of the incoming ACKs. So, after sending 2 segments and receiving the corresponding ACKs, 4 segments are transmitted. When the same 2 segments are sent to a delayed ACK receiver, a single ACK is returned. In response, the sender slides *cwnd* by 2 segments and increments *cwnd* by 1 segment. Therefore, a delayed ACK triggers a burst of 3 back-to-back segments during slow start. Hence, with delayed ACKs the burst in response to each ACK is larger than when the receiver ACKs each incoming segment. However, delayed ACKs reduce the total number of segments sent from 4 segments to 3 segments, in the above example.

2.2 Congestion Avoidance

Congestion avoidance [JK88] [Ste97] [SAP98] is used to probe the network for additional capacity after congestion is detected. The congestion avoidance algorithm increases *cwnd* more conservatively than the slow start algorithm. For each incoming ACK, *cwnd* is increased by $1/cwnd$ (up to the advertised window). When the receiver ACKs each incoming segment, congestion avoidance increases *cwnd* by roughly 1 segment per RTT. However, if delayed ACKs are employed by the receiver only half as many ACKs arrive at the sender and therefore congestion avoidance will increase *cwnd* by roughly half a segment every RTT. Therefore, as with slow start, congestion avoidance induced *cwnd* growth is slowed by delayed ACKs.

2.3 Delayed ACKs After Slow Start

We modified the simulated TCP receivers to send an ACK for each incoming segment while the slow start algorithm is used by the TCP sender. When not using slow start, delayed ACKs are generated. Using *Delayed ACKs After Slow Start (DAASS)* provides more acknowledgments than delayed ACKs during slow start and therefore *cwnd* is opened more rapidly. During the congestion avoidance phase, delayed ACKs are used to conserve host and network resources, without drastically reducing performance.

Using DAASS requires the TCP receiver to determine when the sender is using slow start. One method of accomplishing this may be for the TCP sender to explicitly inform the receiver that delayed ACKs should not be generated. Alternatively, the TCP receiver could use a heuristic to guess when the sender is using the slow start algorithm based on the arrival pattern of incoming data segments. This paper only examines the impact of this mechanism. Researching an implementation of this mechanism is left as future work. Therefore, we cheated and added a global variable in the simulator that indicates whether or not the sender is using slow start.

2.4 Byte Counting

We modified the TCP senders in the simulator to increase *cwnd* based on the number of previously unacknowledged bytes covered by each incoming ACK, rather than the number of ACKs received. This mechanism decouples the sender and receiver behavior and provides the same increase in *cwnd* regardless of how often the receiver generates ACKs. Therefore, the receiver can conserve resources by generating delayed ACKs without impacting the sender's *cwnd* growth. This mechanism will be called *unlimited byte counting (UBC)* for the remainder of the paper. Unlimited byte counting has been shown to improve transfer time in limited tests over satellite links [All97].

As will be shown, unlimited byte counting is too aggressive in some circumstances. Therefore, we investigated a *limited byte counting (LBC)* mechanism, as well. LBC also increases *cwnd* based on the number of previously unacknowledged bytes covered by each incoming ACK. However, the *cwnd* increase can be no more than 2 segments. As will be shown, this limitation on the size of the line-rate burst that an ACK can trigger prevents loss and improves performance when compared to unlimited byte counting.

Paxson [Pax97] found that it is not uncommon for TCP implementations to generate *stretch ACKs*[5] (ACKs for more than two segments). The implications of stretch ACKs are discussed in [Pax97] and [PAD+98]. LBC limits the size of the burst a sender can transmit in response to a stretch ACK when compared to UBC. However, when compared to the standard slow start algorithm LBC increases the size of the burst sent in response to a stretch ACK by 1 segment.

3 Single Flow Tests

This section investigates the performance of the various ACK generation and utilization mechanisms in a simple environment with a single active TCP connection.

3.1 Simulated Environment

We used the *ns* [MF95] network simulator (version 1.4) to conduct our investigation. The Reno and SACK versions of TCP included with *ns* were used for the baseline measurements (TCP with and without delayed ACKs). We made several small changes to the simulator's Reno and SACK TCP implementations to provide the alternate ACK generation and utilization mechanisms studied in this paper.

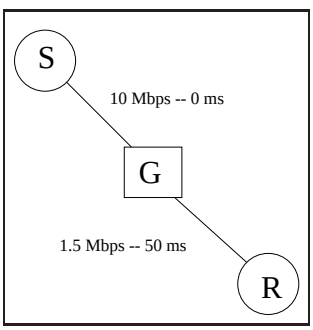


Figure 1: Simulated network topology.

The topology of the network studied is given in figure 1. Labeled in this figure are the data source, S, the data receiver, R, and an intervening gateway, G. As shown in the figure, the bandwidth of the bottleneck link is 1.5 Mbps[6], which is roughly T1 bandwidth. The RTT provided by the simulated network is 100 ms, which is approximately the RTT measured between NASA's Lewis Research Center and the University of California at Berkeley when the topology was constructed.

The bottleneck gateway in the simulated topology utilizes a drop-tail queueing strategy. The TCP data senders use a segment size of 1000 bytes[7]. The advertised window was 40 segments, which is roughly twice the size required to fully utilize the network (the *delay*bandwidth* product).

The size of the queue in the bottleneck gateway varied from 4 segments (heavy loss) to 22 segments (no loss) in our simulations. In addition, the transfer size was varied to show the impact of the alternate ACK generation/utilization mechanisms on various types of traffic (from short WWW pages or email messages to long bulk transfers). The transfer size varied from 5,000 bytes to 995,000 bytes in increments of 10,000 bytes.

3.2 Baseline Measurements

The following is a discussion of TCP's standard acknowledgment generation and utilization strategies [JK88] [Ste97] [SAP98] [Bra89]. These simulations will be used as a baseline with which to compare the alternate ACK generation and utilization mechanisms that will be presented in section 3.3.

3.2.1 Acknowledging Every Segment

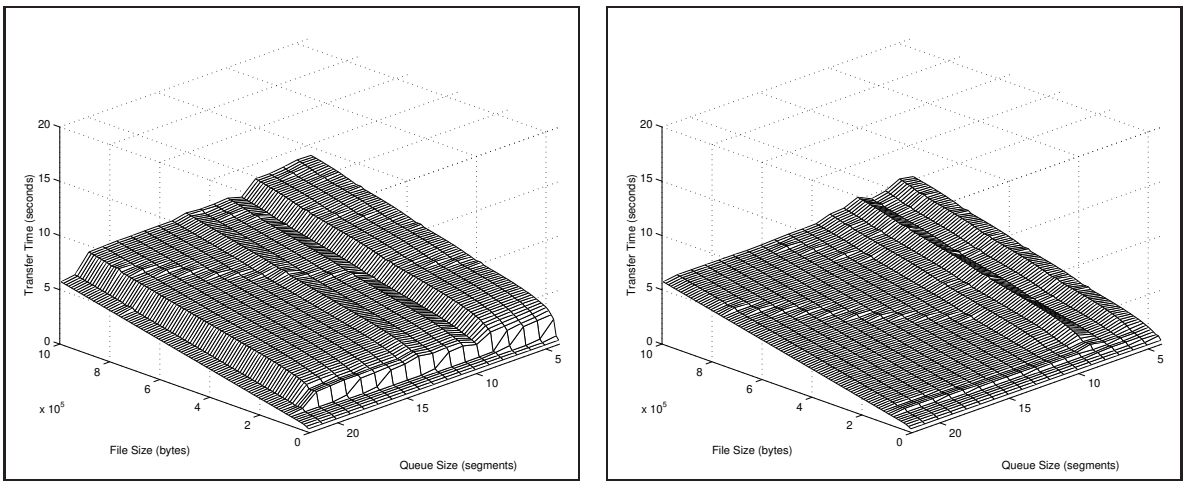
Figure 2(a) shows TCP Reno's behavior when the receiver generates an ACK for each incoming segment, as a function of both bottleneck queue size and transfer size. With a queue size of 22 segments, no loss occurs. When the queue size is reduced to 21 segments, no more than a single segment is dropped from a window of data in any of the transfers. This does not significantly increase the transfer time, as a single loss is repaired using the fast retransmit algorithm without returning to slow start. When the queue size is less than 21 segments, most of the transfers experienced multiple dropped segments in at least one window of data. The multiple drops are caused by the initial slow start phase increasing *cwnd* enough to overwhelm the bottleneck queue. The large increase in transfer time shown when the queue is less than 21 segments represents a TCP retransmission timeout (RTO).

Several short transfers do not experience multiple drops, as shown by their relatively low transfer time. Figure 2(a) illustrates that as the queue size gets smaller, the number of transfers that experience multiple losses increases. This indicates that the loss

[5] It should be noted that buggy TCP implementations are not the only reason for stretch ACKs. A dropped ACK can cause the subsequent ACK to look like a stretch ACK.

[6] In this paper, 1 Mb = 1,000,000 bits.

[7] 1000 bytes is the default segment size in *ns*-1.4



(a) TCP Reno

(b) TCP SACK

Figure 2: Standard TCP sender with receiver ACKing every segment.

is caused by the slow start algorithm overwhelming the queue at the bottleneck. As the size of the queue is reduced, the size of the transfer needed to increase *cwnd* enough to overwhelm the queue and cause multiple drops also decreases.

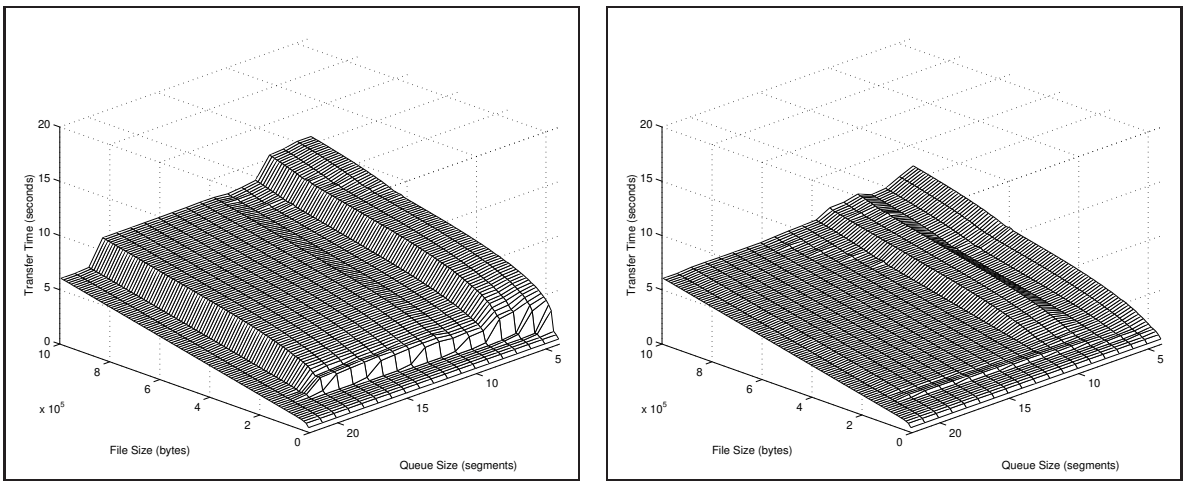
A noticeable increase in transfer time is shown when the queue size is 8 or fewer buffers. This is caused by loss occurring in two successive windows of data. When the queue size is 8 segments, one segment is dropped from the first of the two windows of data and a number of segments are dropped from the subsequent window of data. However, when the queue size is at least 9 segments loss only occurs in the second of the two windows of data. When two successive windows of data experience loss, *cwnd* is reduced twice. Also, the multiple drops from the second window of data require the RTO to expire before retransmitting the lost segments. These two factors lead to an increased transfer time.

Figure 2(b) illustrates the behavior of TCP SACK when the receiver acknowledges each incoming segment. This figure shows no large jumps in transfer time, as shown when using TCP Reno. TCP SACK does not depend on the RTO timer to expire before recovering from multiple dropped segments and therefore is able to repair loss more quickly. The small ripples shown in this graph are caused by small interactions between *cwnd* and the queue size. For instance, the transfer time is generally greater when the queue size is 8 segments when compared to using a queue size of 9 segments. When the queue size is 8 segments, a single segment is lost during loss recovery. However, when the queue size is 9 segments the queue is able to hold this segment, thus preventing a second halving of *cwnd*.

3.2.2 Delayed Acknowledgments

Figure 3(a) shows the behavior of a TCP Reno sender when the receiver employs delayed ACKs. Again, when the queue size is 22 segments no loss occurs. In this case, no loss occurs when the queue is 21 segments due to the more gradual increase of *cwnd* caused by delayed ACKs. When the queue size is 20 segments no more than a single drop occurs in any window of data. This single loss is repaired using the fast retransmit algorithm and therefore does not cause a large increase in transfer time. In addition, this figure again shows that short transfers do not experience loss because TCP is unable to open *cwnd* enough to overwhelm the queue before the transfer is complete. However, the transfer size required to overwhelm the queue is generally greater when the receiver uses delayed ACKs than when the receiver ACKs every segment. This is explained by the increased time required to open *cwnd* in the face of delayed ACKs, as discussed in section 2.1.

As in the previous section, the increase in the transfer time for most transfer sizes when the queue size is 19 segments can be explained by slow start overwhelming the bottleneck queue causing multiple dropped segments from a window of data. This causes TCP Reno to incur an RTO timeout and resort to slow start to retransmit segments. The in-



(a) TCP Reno

(b) TCP SACK

Figure 3: Standard TCP sender with a receiver generating delayed ACKs.

crease in transfer time when the queue size is 19 segments is greater when the receiver generates delayed ACKs than when the receiver ACKs each incoming segment. The increase is explained by the increase in time required by the slow start and congestion avoidance algorithms to raise *cwnd* after loss when using delayed ACKs (as discussed in sections 2.1 and 2.2). The jump in the transfer time when the queue size is 7 or fewer segments is caused by dropped segments in two successive windows of data, as discussed in section 3.2.1. Note that the increase happens when the queue size is 7 segments when using delayed ACKs, rather than 8 segments, as happens when the receiver ACKs each incoming segment. Again, this is caused by *cwnd* increasing more slowly when delayed ACKs are generated.

Figure 3(b) shows the behavior of TCP SACK when the receiver generates delayed acknowledgments. This figure shows no large jumps in transfer time, as shown in the TCP Reno transfers. As in the case when the TCP SACK receiver ACKs each segment (section 3.2.1), the small ripples in this figure are caused by interactions between the queue size and *cwnd*. The ripples are slightly larger when the receiver employs delayed ACKs because the congestion avoidance algorithm takes roughly twice as long to increase *cwnd* after the recovery phase when delayed ACKs are used.

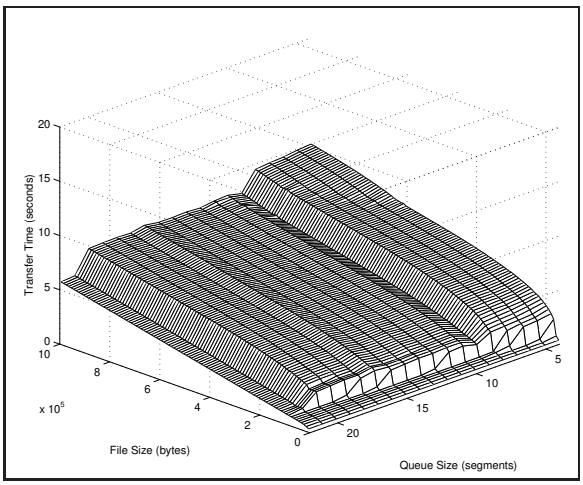
3.3 Alternative Mechanisms

3.3.1 Delayed ACKs After Slow Start

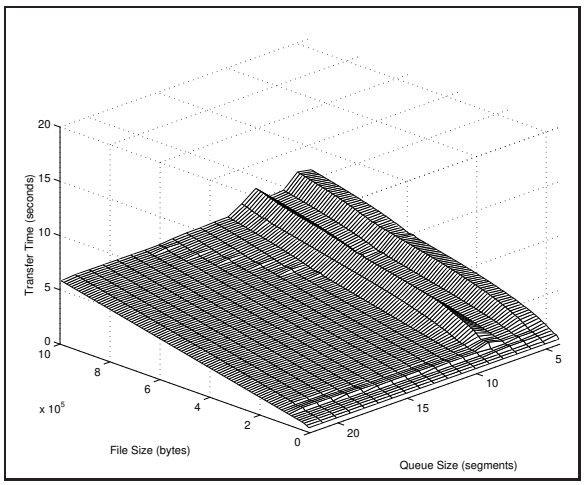
Figure 4(a) illustrates the behavior of TCP Reno when the receiver uses DAASS. In this case, the behavior of TCP Reno is similar to that shown when using the standard acknowledgment mechanisms (sections 3.2.1 and 3.2.2). Figure 4(b) shows the behavior of TCP SACK when the receiver uses DAASS. This figure shows small increases in transfer time at various points. When the queue size is 8 segments, the transfer time increases. As indicated in sections 3.2.1 and 3.2.2 this is an interaction between the queue size and *cwnd*. An additional increase in transfer time is shown when the queue size is 5 segments. This happens because the queue size causes the transfer to experience loss during the recovery phase. This loss further reduces *cwnd*, increasing the transfer time.

3.3.2 Unlimited Byte Counting

Figure 5(a) shows the behavior of TCP Reno using unlimited byte counting (UBC) in the face of delayed ACKs from the receiver. This figure shows large variations in the transfer time. These increases in transfer time are caused by large increases in *cwnd* during loss recovery. Acknowledgments covering a large number of previously unacknowledged segments are commonly received during the slow start phase following a loss event. When using UBC, an ACK covering $\mathcal{N}$ previously unacknowledged segments triggers a burst of $2\mathcal{N}$ new segments to be sent. In the case

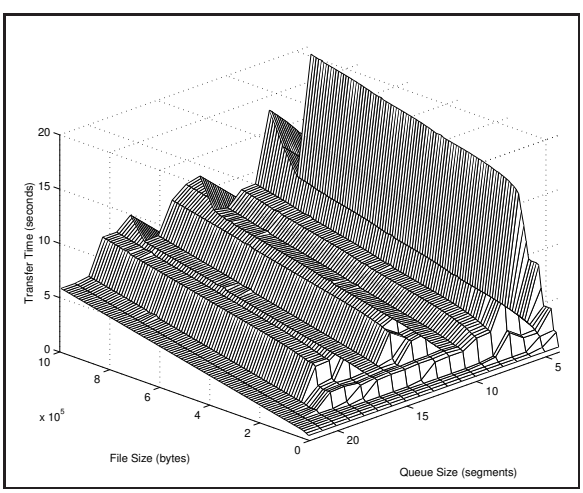


(a) TCP Reno

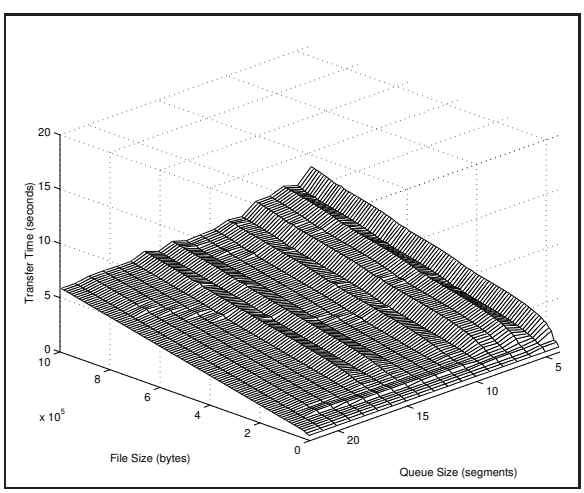


(b) TCP SACK

Figure 4: Standard TCP sender with receiver generating delayed ACKs after slow start.



(a) TCP Reno



(b) TCP SACK

Figure 5: Sender using unlimited byte counting with a receiver generating delayed ACKs.

when $\mathcal{N}$ is large, the resulting line-rate burst of segments may overwhelm the bottleneck queue causing further loss and consequently an increase in transfer time, as shown.

Figure 5(b) shows the behavior of TCP SACK using UBC when the receiver is generating delayed ACKs. The variation in the transfer time in this figure is relatively small when compared to UBC under TCP Reno. Generally, TCP SACK does not revert to slow start during recovery and keeps the network (and *cwnd*) filled by injecting new segments during recovery. Therefore, cumulative ACKs that cover a large number of previously unacknowledged segments do not represent permission to send a large burst of new data segments, which could cause further loss. UBC does lead to larger line-rate bursts in the face of delayed ACKs than when the receiver ACKs each segment. Without UBC, bursts of 3 segments during slow start are normal because receipt of a delayed ACK slides *cwnd* by 2 segments and slow start increments *cwnd* by an additional segment. UBC increments *cwnd* by 2 additional segments upon receipt of a delayed ACK, which leads to a 4 segment burst. This added burstiness causes a small amount of additional congestion, leading to the the small variation in transfer time shown in figure 5(b).

3.3.3 Limited Byte Counting

To reduce the burstiness of UBC during TCP Reno style loss recovery we added a limit of 2 segments to the amount *cwnd* can be incremented by an incoming ACK. Limiting the increase of *cwnd* to 2 segments will allow TCP senders to open *cwnd* faster than using the standard increment algorithm when the receiver is properly generating delayed ACKs. However, overly excessive bursts caused by stretch ACKs or slow start after loss can be avoided.

Figure 6 shows the behavior of TCP Reno with limited byte counting (LBC) in the face of delayed ACKs. This figure shows that the 2 segment limit prevents inappropriate burstiness during loss recovery, which in turn prevents further loss as was evident with UBC. The increase in transfer time that occurs when the queue size is 7–9 segments is caused by a small interaction between *cwnd* and the queue size. When the queue size is 7–9 segments, enough new segments are sent during recovery (with the fast recovery algorithm) to generate enough ACKs to trigger the fast retransmit algorithm for two segments dropped within one window of data. This successive fast retransmit phenomenon [Flo95], causes the RTO timer to reset, causing a large idle period before using slow start to retransmit the remaining dropped segments.

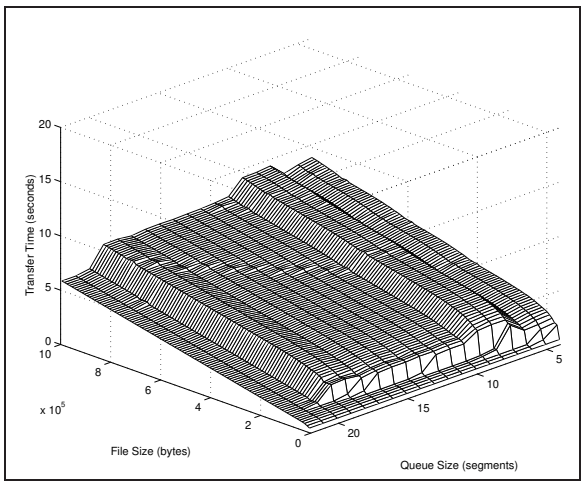


Figure 6: Sender using TCP Reno and limited byte counting with a receiver generating delayed ACKs.

When the queue size is 7–9 segments the sender is able to inject enough new segments into the network using fast recovery to allow this phenomenon to occur. However, when the queue size is less than 7 segments, more segments are lost reducing the number of duplicate ACKs generated by the receiver. This prevents the sender from injecting enough new data into the network to cause the second fast retransmit and therefore slow start recovery occurs sooner. The loss pattern that occurs when the queue size is 10 segments provides no possibility of a successive fast retransmit within a window of data, allowing slow start to begin sooner, thus reducing the transfer time.

The behavior of TCP SACK using LBC is exactly the same as when using UBC (figure 5(b)). As discussed in section 3.3.2, TCP SACK recovery does not generate large cumulative ACKs that lead to large bursts of segments that can cause additional loss. Therefore, transfer time is not changed by the placing a limit on the byte counting algorithm when using TCP SACK. However, placing a limit on byte counting is still important, even when using TCP SACK. The limit guards against stretch ACKs (caused by buggy TCP implementations or dropped ACKs) triggering inappropriately large bursts of traffic that may overwhelm the bottleneck queue.

3.4 Comparisons

Figure 7 shows the transfer time as a function of transfer size for the various mechanisms studied in this paper. Note that only short transfers are shown in the plot, but the curves continue as shown for larger transfer sizes. The queue size for these trans-

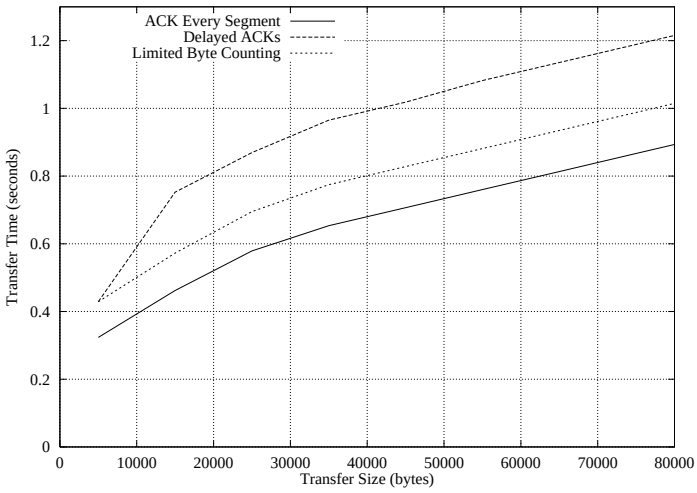


Figure 7: Loss-free transfer time comparison.

fers was 22 segment buffers and therefore no loss occurred. Therefore, TCP Reno and TCP SACK exhibit the same behavior. UBC and LBC exhibit the same behavior in the absence of segment drops so only the LBC transfers are plotted. The transfer time for the case when the standard *cwnd* increase algorithm is used in the face of a receiver ACKing each segment and DAASS is identical and therefore, DAASS is not plotted. The two algorithms perform exactly the same in the absence of loss since all *cwnd* growth is provided during slow start and both algorithms receive the same number of ACKs during slow start. DAASS provides slower *cwnd* growth after loss (i.e., during congestion avoidance) due to the reduced number of ACKs being generated by the receiver.

Using the standard *cwnd* increase algorithm in the face of ACKing every segment provides the best performance. Using delayed ACKs with the standard *cwnd* increase algorithm provides the worst performance. The delayed ACK timer prevents LBC/UBC from obtaining the same performance as the case when the receiver ACKs each incoming segment. However, LBC/UBC still provide better performance than using the standard *cwnd* increase algorithm with delayed ACKs. The difference between LBC/UBC and standard TCP with ACKing every segment is explained by the delayed ACK timeout required to ACK the first segment sent in the transfer. We repeated the simulation with a 2 segment initial congestion window, as proposed in [AFP98], and the results show that LBC/UBC performance was nearly identical to standard TCP when the receiver ACKs each incoming segment.

Figure 8 shows the percentage of time in which the queue at the bottleneck link was a given length. As described above, the maximum queue size was 22 segments in this simulation and no loss occurred in any of the transfers. As the figure shows, a queue length of 20 segments dominated when the receiver ACKed each incoming segment. All other mechanisms use delayed ACKs and the queue spent the vast majority of the time with a length of 19 or 20 segments. In the case of delayed ACKs, new segments are generated roughly half as often as when each segment is acknowledged. The larger period of time between ACKs allows the router to transmit a segment and therefore the queue size oscillates between the two queue lengths. This figure shows that all the mechanisms utilize the queue in a similar manner.

Table 1 presents the median percent improvement in TCP Reno transfer time using various ACK generation/utilization mechanisms when compared to a standard TCP Reno sender and a receiver generating delayed ACKs. In the top half of the table all transfer sizes reported in figures 2(a) – 6 are considered. In the bottom half of the table, the improvement for "short" transfers is reported. A "short" transfer is defined as all transfers reported in figures 2(a) – 6 that do not exceed 80,000 bytes. In our simulations, a loss-free transfer under 80,000 bytes uses the slow start algorithm for the entire transfer. However, a transfer of more than 80,000 bytes is able to complete the slow start phase. Table 2 presents a similar comparison for a TCP SACK sender and a receiver generating delayed ACKs and SACK blocks.

Acknowledging each segment provides the best improvement in transfer time in all conditions presented in the tables. However, ACKing each segment gener-

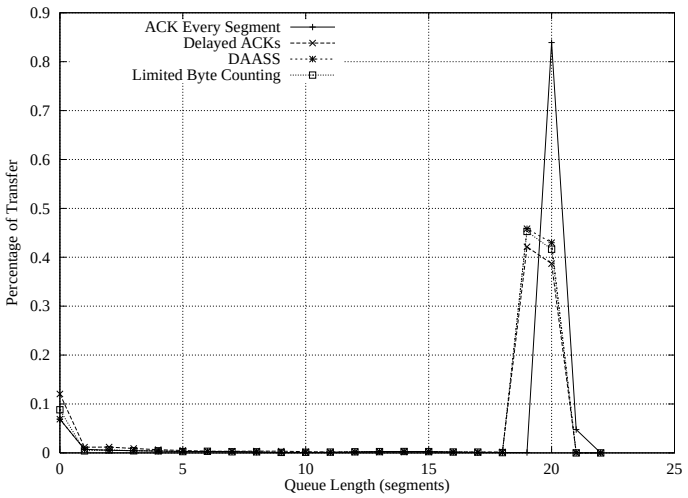


Figure 8: Bottleneck queue size.

Experimental Mechanism	**Transfers**	**Delayed ACKs** (Median % Improvement)
ACK Each Segment	All	12
Delayed ACKs After Slow Start	All	7
Unlimited Byte Counting	All	-9
Limited Byte Counting	All	8
ACK Each Segment	Short	28
Delayed ACKs After Slow Start	Short	28
Unlimited Byte Counting	Short	18
Limited Byte Counting	Short	19

Table 1: Reno Comparison

Experimental Mechanism	**Transfers**	**Delayed ACKs** (Median % Improvement)
ACK Each Segment	All	9
Delayed ACKs After Slow Start	All	5
Unlimited Byte Counting	All	3
Limited Byte Counting	All	3
ACK Each Segment	Short	28
Delayed ACKs After Slow Start	Short	20
Unlimited Byte Counting	Short	17
Limited Byte Counting	Short	17

Table 2: SACK Comparison

ates more segments which can waste scarce host and network resources. Using DAASS improves transfer time when compared to standard TCP with delayed ACKs in all cases shown in the tables. However, the mechanism also consumes more host and network resources than standard TCP with delayed ACKs during slow start. As expected, table 1 shows that acknowledging every segment all the time is roughly the same as DAASS for short transfers that are dominated by slow start (either at connection startup or during loss recovery). Acknowledging every segment performs better than DAASS for short transfers using SACK (table 2) because SACK does not revert to slow start to repair multiple dropped segments. Therefore, since DAASS is less aggressive during congestion avoidance *cwnd* does not grow as rapidly as when an ACK is generated for each incoming segment and consequently the performance is not as good.

As discussed in section 3.3.2, TCP Reno using UBC increases the transfer time when considering all transfers, compared to standard TCP Reno in the face of delayed ACKs. However, when only short transfers are considered, UBC reduces the transfer time as shown in the tables. Short transfers do not experience as much loss as longer transfers, thus for short transfers the problems with UBC do not come into play as often. As discussed in section 3.3.2, TCP SACK does not have the same problems as TCP Reno when using UBC. Therefore UBC provides modest improvements in performance when TCP SACK is employed.

Tables 1 and 2 show that LBC performs better than the standard *cwnd* increase algorithm in the face of delayed ACKs. The performance benefit is more dramatic in short transfers that are dominated by slow start. LBC does not perform as well as the standard *cwnd* increase algorithm when the receiver ACKs each incoming segment. However, limited byte counting does increase performance without increasing the number of segments injected into the network.

4 Competition Tests

This section provides a discussion of the various acknowledgment generation and usage mechanisms in the presence of competing traffic. The simulation setup is given followed by a comparison of the various mechanisms in a drop-tail queueing environment and a comparison in a RED queueing environment.

4.1 Simulated Environment

The topology for these tests is the same as used in the previous section (figure 1). TCP SACK is used in all tests and all ACK generation and utilization mechanisms except UBC are used. As shown in section 3, UBC hurts the performance of a single TCP connection in many cases and therefore it is not studied in the context of competing flows. The advertised TCP window size is 20 segments, or roughly the delay-bandwidth product of the bottleneck link. The maximum queue length in the gateway is set to 40 segments which is roughly twice the delay-bandwidth product of the bottleneck link. The first set of simulations utilizes a drop-tail queue in the gateway. In the second set of simulations RED queueing [FJ93] [BCC+98] was used at the bottleneck link. Table 3 provides a list of the RED parameters used in these simulations.

Parameter	Value
min_{th}	7
max_{th}	21
w_q	0.002
max_p	0.1

Table 3: RED Settings

The transfer size for each TCP connection is randomly picked between 5 KB[8] and 100 KB. The number of TCP connections started in each simulation is varied from 50 to 500 (in increments of 50 connections). The time at which each transfer starts is picked randomly between 0 seconds (the beginning of the simulation) and 100 seconds. The simulation ended when all transfers completed.

4.2 Drop-Tail Queueing

Figure 9 shows the average per flow throughput as a function of the number of TCP connections completed during the simulation. In this simulation, the router uses a drop-tail queue. This figure shows that using the standard *cwnd* increase algorithm and ACKing every segment provides the best throughput. Additionally, using the standard *cwnd* increase algorithm with delayed ACKs provides the worst throughput. LBC and DAASS provide throughput that is close to standard TCP with ACKing every segment, however LBC generally provides slightly better throughput than DAASS. When the network is under heavy load (400-500 flows during the course of the simulation) all mechanisms exhibit similar performance. In a highly congested environment TCP is only able to utilize a small *cwnd*, so the slow start phase is relatively short. Since the slow start phase is short and the mechanisms being studied mainly help

[8] In this paper, 1 KB is 1,000 bytes.

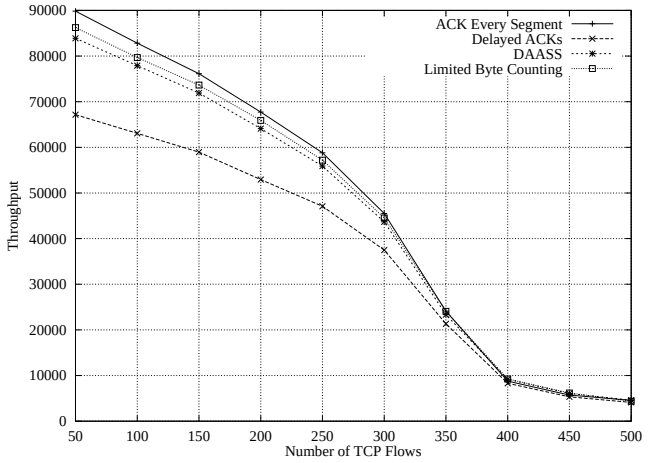


Figure 9: Average Throughput – Drop-Tail Queueing

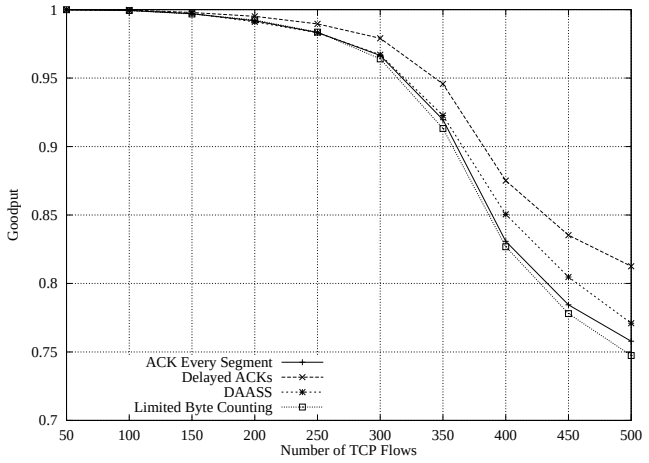


Figure 10: Average Goodput – Drop-Tail Queueing

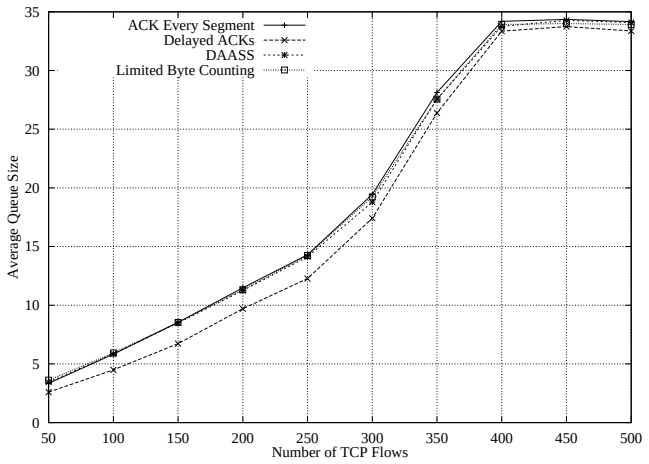


Figure 11: Average Queue Length – Drop-Tail Queueing

slow start it is not surprising that all variants of TCP perform similarly in a highly congested environment.

Figure 10 shows the average goodput[9] as a function of the number of TCP flows completed during the simulation. Using a standard TCP sender with delayed ACKs provides the highest goodput, while using LBC provides the worst goodput. Using delayed ACKs makes the *cwnd* increase less rapid and less bursty which in turn reduces loss. Meanwhile, the other mechanisms studied open *cwnd* more aggressively, which leads to burstier traffic and more loss. DAASS does not experience as much loss as ACKing every segment or using LBC because it opens the congestion window less aggressively during congestion avoidance. LBC causes slightly more loss than standard TCP with ACKing every segment due to the larger line-rate bursts caused by LBC. When receiving an ACK for each segment, a standard TCP sender transmits at most 2 back-to-back segments per ACK. However, when using LBC an ACK can cover 2 segments, allowing LBC senders to transmit up to 4 back-to-back segments per ACK during slow start.

Figure 11 shows the intervening gateway's average queue length as a function of the number of TCP flows in a given simulation. Using delayed ACKs provides a slight reduction in the average queue length when compared to the other mechanisms studied in this paper. The alternate mechanisms studied in this paper provided an average queue length that is very similar to the case when the receiver ACKs each incoming segment.

4.3 RED Queueing

Figure 12 shows the average throughput as a function of the number of TCP flows active during the simulation. In these simulations, the intervening router utilizes RED queueing, as described in section 4.1. Using the standard *cwnd* increase algorithm while ACKing every segment provides the highest throughput, while standard TCP with delayed ACKs yields the lowest throughput. Unlike the drop-tail case, using delayed ACKs has a negative impact on throughput in all simulations, even those consisting of a large number of flows. RED queues are better able to handle small bursts of segments, which allows the more aggressive mechanisms to obtain better throughput than the standard algorithm in response to delayed ACKs. The LBC and DAASS mechanisms were able to obtain throughput close to the standard algorithm with ACKing every segment in all cases. LBC again performs slightly better than DAASS due to less aggressive nature of DAASS during congestion avoidance.

Figure 13 shows average goodput as a function of the number of TCP flows completed during a simulation. This figure shows that using delayed ACKs and TCP's standard *cwnd* increase algorithm provides the best goodput (i.e., the fewest drops). Using the standard *cwnd* increase algorithm coupled with ACKing every incoming segment provides the worst goodput. Using LBC or DAASS provides goodput between the standard algorithm in response to ACKing every segment and delayed ACKs. This figure shows that LBC achieves better goodput than standard TCP when the receiver ACKs each segment. This differs from the results of the drop-tail simulations, where LBC experienced more loss (and therefore, less goodput). This shows that RED is able to handle the small bursts caused by LBC better than drop-tail queueing. Also note the goodput of all mechanisms is less when using RED queueing than under drop-tail queueing (figure 10). The reduction in goodput is caused by the early dropping performed by RED.

Although the loss rate is increased slightly when using RED queueing, the average queue size is reduced. Figure 14 shows the average RED queue size as a function of the number of TCP connections completed during the simulation. This figure shows RED is able to manage the average queue size and therefore better able to handle relatively small bursts of segments than a drop-tail queue.

5 Telnet Considerations

The last area of our investigation involves the use of interactive telnet traffic in conjunction with two of the ACK generation/utilization mechanisms investigated in this paper. In this section, we investigate standard TCP with delayed ACKs and limited byte counting. For the purposes of this section, standard TCP with ACKing every segment and DAASS are sufficiently similar to standard TCP with delayed ACKs that they are left out of the discussion.

Figure 15 shows the congestion window as a function of time for a simulated 120 second telnet connection between the sender and receiver shown in figure 1. The tick marks along the x-axis show the transmission time for each segment in the flow. For this plot, the standard *cwnd* increase algorithm was used by the sender and the receiver generated delayed ACKs. The congestion window was able to reach the advertised window within the first 30 seconds of the transfer. However, the actual number of outstanding packets was roughly 2–3 at any given time. Allowing *cwnd* to grow as shown in the figure can cause large bursts of

[9] We define goodput as the ratio of the number of unique data bytes sent to the number of total data bytes sent.

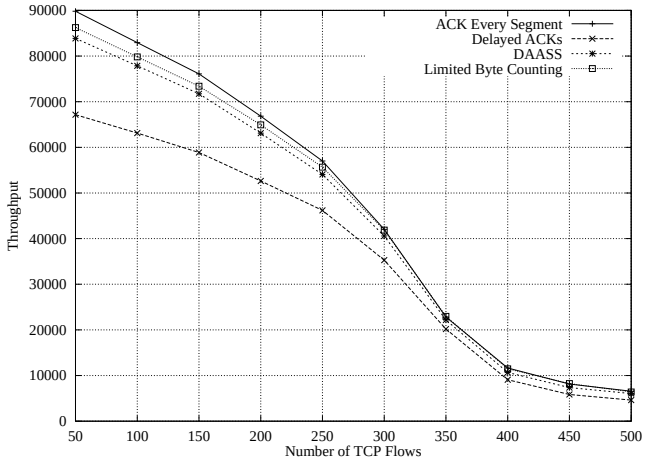


Figure 12: Average Throughput – RED Queueing

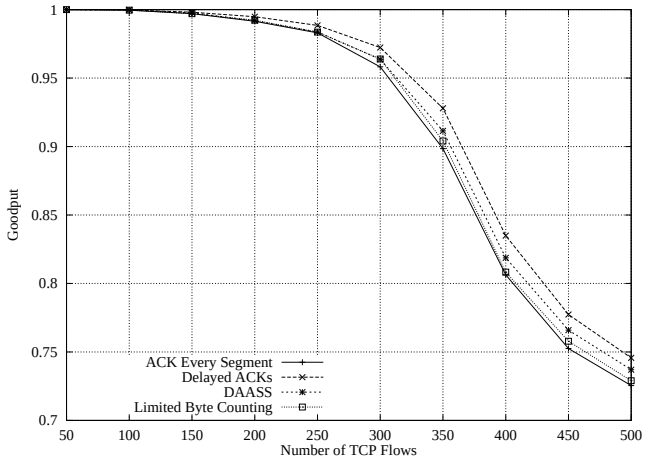


Figure 13: Average Goodput – RED Queueing

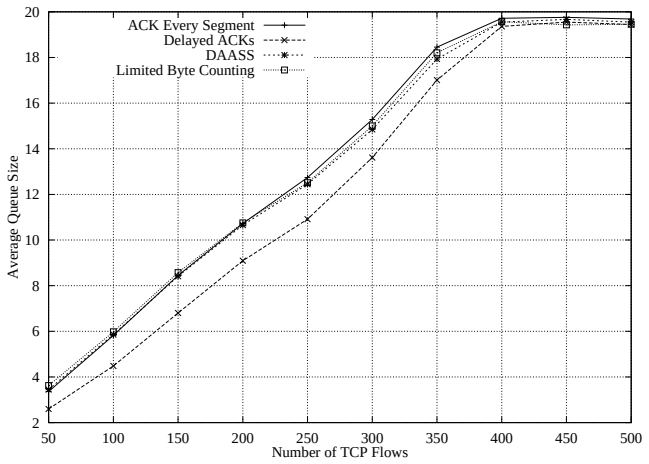


Figure 14: Average Queue Length – RED Queueing

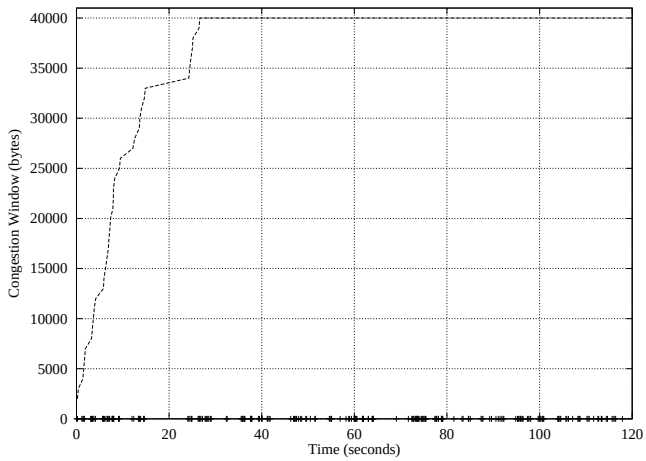


Figure 15: Telnet Connection – Delayed ACKs

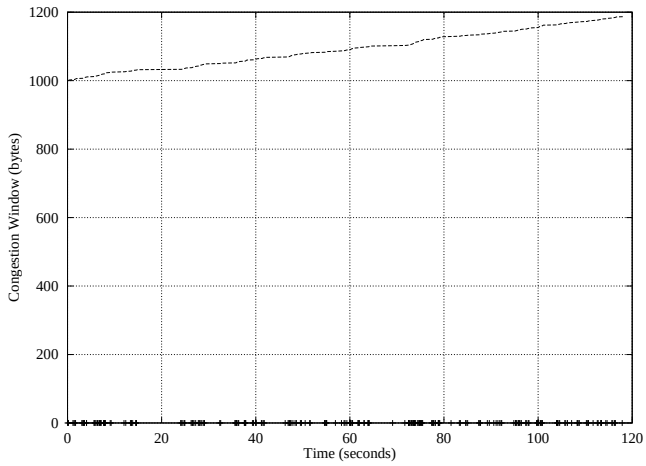


Figure 16: Telnet Connection – Limited Byte Counting

traffic. For example, if a user, who has typed enough to fully open *cwnd*, suddenly displays a large file over the connection, a large line-rate burst of network traffic occurs. This burst may be inappropriate for the current network conditions and may cause loss.

We repeated the above simulation employing LBC at the sender. As shown in figure 16, *cwnd* continues to increase over the life of the connection. However, the increase is rather modest given that the increase in *cwnd* is based on the small number of bytes ACKed by each incoming acknowledgment, rather than blindly incrementing *cwnd* by one segment per ACK. While *cwnd* continues to grow, telnet is not as susceptible to generating bursty traffic as if the standard algorithm is used.

6 Conclusions and Future Work

This paper has presented simulation investigations of Reno and SACK TCP using both standard ACKing mechanisms and three alternate methods of generating and utilizing acknowledgments. Below are some of our key findings.

- ACKing each segment provides the best performance across all simulations presented in this paper. The cost of ACKing each incoming segment, when compared to using delayed ACKs, is a slight increase in the loss rate, caused by more aggressive congestion window growth. Changing TCP implementations to ACK each segment will also increase the number of segments transmitted into the network. Whether or not it is safe or wise to increase the number of ACKs generated in large shared networks is an open question.

- ACKing each segment only during slow start improves performance over standard TCP with delayed ACKs. Using DAASS also modestly increases the segment drop rate due to the increased aggressiveness during slow start. As with ACKing each segment, DAASS increases the number of ACKs inserted into the network and therefore must be carefully considered. Finally, implementing DAASS requires further investigation into a mechanism by which the receiver can determine when the sender is using the slow start algorithm.

- Unlimited byte counting in conjunction with TCP Reno drastically increases burstiness during loss recovery and many times causes additional loss. This in turn produces a negative impact on performance. Therefore, *unlimited byte counting is not recommended.*

- Limited byte counting provides performance improvement across all scenarios presented in this paper. In addition, this mechanism is easy to implement and provides performance gains comparable to using standard TCP and allowing the receiver to ACK each incoming segment. The cost of limited byte counting is a slight increase in burstiness and loss. As shown in section 4.2, LBC increases the drop rate slightly compared to standard TCP when receiving an ACK for each segment in networks utilizing drop-tail queues. However, section 4.3 shows that when RED queueing is utilized, the additional burstiness caused by LBC is more easily absorbed by the gateway. Furthermore, LBC achieves added performance without increasing the number of ACKs the receiver is required to generate. While LBC is promising, the mechanism requires further study in real networks before it can be recommended for wide-spread use.

- LBC provides more appropriate congestion window growth for interactive applications. We believe that *LBC should be used if an incoming ACK covers less than one segment size of outstanding data.*

- We also feel that *unlimited* byte counting *may* be appropriate for future versions of TCP. When coupled with SACK-based loss recovery UBC is no worse than LBC. Also, topologies exist where an ACK interval of more than 2 segments is appropriate based on the ratio of the bandwidth in the forward direction to the bandwidth of the return path [BPK97]. Several researchers have investigated using an ACK interval larger than 2, as well as varying the ACK interval dynamically [Joh95] [BPK97]. However, using a larger ACK interval provides fewer ACKs which in turn can hurt performance on the forward channel. Therefore, unlimited byte counting is a promising approach to decoupling TCP's behavior on the forward path from the behavior on the return path. However, using a large ACK interval and UBC can produce very large line-rate bursts. To alleviate these large bursts of data packets a *segment pacing*[10] algorithm will likely be required.

[10] TCP pacing after a long idle period is discussed in [VH97]. An indirect method of pacing segments (by spacing incoming ACKs) is discussed in [Par98]. In addition, Van Jacobson outlined a general method for TCP pacing in the TCP Implementations and TCP Over Satellite Working Group meetings at the Munich IETF, August 1997.

A TCP with the characteristics described above still requires much more research.

Acknowledgments

This paper benefited from the help of a number of people. Thanks to Sally Floyd, Dan Glover, Chris Hayes, Matt Mathis, Jeff Semke and Tim Shepard for discussions about the mechanisms investigated in this paper. Jim Griner provided help with the script used to analyze the simulation data. Will Ivancic, Paul Mallasch, Jeff Semke and the anonymous reviewers (both CCR and SIGCOMM 98) provided valuable feedback on the paper. My thanks to all.

References

[AFP98] Mark Allman, Sally Floyd, and Craig Partridge. Increasing TCP's Initial Window, September 1998. RFC 2414.

[All97] Mark Allman. Improving TCP Performance Over Satellite Channels. Master's thesis, Ohio University, June 1997.

[BCC+98] Robert Braden, David Clark, Jon Crowcroft, Bruce Davie, Steve Deering, Deborah Estrin, Sally Floyd, Van Jacobson, Greg Minshall, Craig Partridge, Larry Peterson, K. Ramakrishnan, S. Shenker, J. Wroclawski, and Lixia Zhang. Recommendations on Queue Management and Congestion Avoidance in the Internet, April 1998. RFC 2309.

[BPK97] Hari Balakrishnan, Vakata Padmanabhan, and Randy Katz. The Effects of Asymmetry on TCP Performance. In *ACM MobiCom*, September 1997.

[Bra89] Robert Braden. Requirements for Internet Hosts – Communication Layers, October 1989. RFC 1122.

[FF96] Kevin Fall and Sally Floyd. Simulation-based Comparisons of Tahoe, Reno, and SACK TCP. *Computer Communications Review*, 26(3), July 1996.

[FF98] Sally Floyd and Kevin Fall. Promoting the Use of End-to-End Congestion Control in the Internet. Technical report, LBL, February 1998.

[FJ93] Sally Floyd and Van Jacobson. Random Early Detection Gateways for Congestion Avoidance. *IEEE/ACM Transactions on Networking*, 1(4):397–413, August 1993.

[Flo95] Sally Floyd. TCP and Successive Fast Retransmits. Technical report, Lawrence Berkeley Laboratory, May 1995.

[Jac90] Van Jacobson. Modified TCP Congestion Avoidance Algorithm. Technical report, LBL, April 1990. Email to the end2end-interest mailing list. URL: ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[JK88] Van Jacobson and Michael Karels. Congestion Avoidance and Control. In *ACM SIGCOMM*, 1988.

[Joh95] Stacy Johnson. Increasing TCP Throughput by Using an Extended Acknowledgement Interval. Master's thesis, Ohio University, June 1995.

[MF95] Steven McCanne and Sally Floyd. NS (Network Simulator), 1995. URL http://www-nrg.ee.lbl.gov.

[MMFR96] Matt Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. TCP Selective Acknowledgement Options, October 1996. RFC 2018.

[PAD+98] Vern Paxson, Mark Allman, Scott Dawson, Jim Griner, Ian Heavens, Kevin Lahey, Jeff Semke, and Bernie Volz. Known TCP Implementation Problems, August 1998. Internet-Draft draft-ietf-tcpimpl-prob-04.txt (work in progress).

[Par98] Craig Partridge. ACK Spacing for High Delay-Bandwidth Paths with Insufficient Buffering, August 1998. Internet-Draft draft-rfced-info-partridge-00.txt (work in progress).

[Pax97] Vern Paxson. Automated Packet Trace Analysis of TCP Implementations. In *ACM SIGCOMM*, September 1997.

[Pos81] Jon Postel. Transmission Control Protocol, September 1981. RFC 793.

[PS97] Craig Partridge and Tim Shepard. TCP/IP Performance Over Satellite Links. *IEEE Network*, 11(5), September/October 1997.

[SAP98] W. Richard Stevens, Mark Allman, and Vern Paxson. TCP Congestion Control, August 1998. Internet-Draft draft-ietf-tcpimpl-cong-control-00.txt (work in progress).

[Ste97] W. Richard Stevens. TCP Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery Algorithms, January 1997. RFC 2001.

[VH97] Vikram Visweswaraiah and John Heidemann. Improving Restart of Idle TCP Connections. Technical Report 97-661, University of Southern California, August 1997.