It has long been an axiom of mine that the little things are infinitely the most important.

— Sherlock Holmes, "A Case of Identity," by Sir Arthur Conan Doyle

Chapter 7

Enhanced Loss Recovery for Small Windows

In previous chapters, we addressed the problems caused by wireless bit-errors and asymmetric effects on TCP performance and solved them using a combination of link-layer and end-to-end techniques. In this chapter, we analyze the problems that arise due to small TCP transmission windows and solve them by enhancing TCP's data-driven loss recovery mechanism. Small transmission windows are the norm in many wireless networks that have inherently low channel bandwidths. Because the bandwidth-delay product of the connection is a measure of its sustainable transmission window, low channel bandwidths imply small windows. In addition, short transfers typical of today's Web workload add to the prevalence of small windows.

The result of small TCP windows is that data-driven loss recovery using fast retransmissions is not triggered often; most losses require expensive timeouts to recover from, keeping the connection idle for long periods of time. Our goal is to incur expensive timeouts only when persistent congestion occurs and not when transient losses occur in the network. By designing enhancements to TCP's loss recovery algorithms to make it more data-driven, we reduce the reliance on conservative retransmission timers.

The rest of this chapter is organized as follows. We start by analyzing packet-level traces of over 1.6 million TCP connections collected from the IBM-run official Web server of the 1996 Atlanta Summer Olympic Games [104] to understand the problems caused by small windows. Then, after discussing the limitations of TCP selective acknowledgments (SACK) [92] in solving these problems, we present the details of our loss recovery enhancements to TCP. We conclude the chapter by discussing simulation and implementation performance over wireless links and cross-country Internet transfers.

7.1 Analysis Overview

To understand the behavior of TCP connections in today's Internet, we analyze packet-level traces of connections to a busy Web server. In particular, we seek to answer the following questions:

1. *Loss recovery:* Do current TCP senders recover from packet losses using data-driven loss recovery mechanisms such as fast retransmission and fast recovery or do they tend to incur coarse timeouts when losses occur?
2. *Selective Acknowledgments (SACKs):* What are the potential benefits of using more sophisticated loss recovery mechanisms such as SACKs? How beneficial are SACKs likely to be in reducing the frequency of timeouts?
3. *Receiver bottleneck:* How often do receiver-advertised windows, used to perform flow control, restrict the network performance of a connection? Are the effective window sizes available to senders often limited by small receiver-advertised window sizes?

To answer these questions, we analyze a very large data set—packet-level traces of over 1.6 million TCP connections collected from the official Web server for the 1996 Atlanta Olympic games. Over a three-week period, the traces included approximately 1.5 billion TCP packets from 700,000 distinct hosts from all over the world.

Our analysis of the traces yield the following answers to the questions raised above:

1. *Existing loss recovery techniques are not effective in recovering from packet losses in a timely manner.* We find that over 50% of all losses are recovered only after a coarse timeout; fast retransmissions recover from less than 45% of all losses. This implies that better solutions are needed to perform efficient loss recovery.

2. *The deployment of SACKs is not likely to significantly reduce the occurrence of timeouts.* In these traces, only a little over 4% of the observed timeouts could have been avoided by the presence of SACKs.

3. *Receiver socket buffer sizes are sometimes too small, making the connection throughput limited by the receiver-advertised window.* Future network implementations should increase their default socket buffer size to avoid the receiver window from becoming a bottleneck on typical Internet paths. The socket buffer size limited the throughput of approximately 14% of all observed connections.

Table 7.1 summarizes the results of our analysis (details in Section 7.4).

Trace Statistic	**Value**	**%**
Total connections	*1,650,103*	*100*
With packet reordering	97,036	6
With receiver window as bottleneck	233,906	14
Total packets	*7,821,638*	*100*
During slow-start	6,662,050	85
# of slow-start pkts lost	354,566	5
During congestion avoidance	1,159,588	15
# of congestion avoidance pkts lost	82,181	7
Total retransmissions	*857,142*	*100*
Fast retransmissions	375,306	44
Slow-start retransmissions	59,811	7
Coarse timeouts retransmissions	422,025	49
Avoidable with SACKs	18,713	4
Avoidable with enhanced recovery	104,287	25

Table 7.1 Summary of analysis (percentages are relative to the category above it).

7.2 Data Collection and Processing

This section describes the details of the trace collection and post-processing we performed on the packet traces. We start with a brief description of the Web server location and configuration.

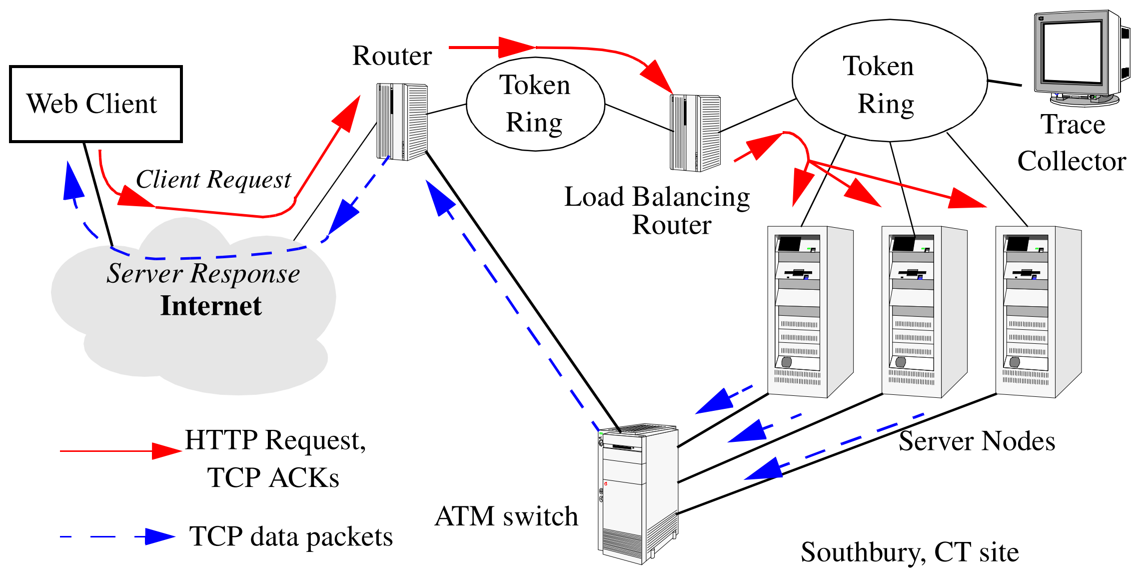

Figure 7.1 WWW server and monitoring setup.

7.2.1 Web Site

The Web traces used in this chapter come from the primary Web site (in Southbury, CT) that IBM ran for the Atlanta Olympic games [104]. Figure 7.1 shows the topology of the Web site's network and the hardware used at the site. During the Olympics, the site was connected via a T3 link to each of the four U.S. Network Access Points (NAPs), located in Chicago (Bellcore and Ameritech), the San Francisco Bay Area (Bellcore and Pacific Bell), New York (Sprint), and Washington, D.C. (MFS Datanet). In addition, there were mirror sites at Cornell (N.Y.), Keio (Japan), Karlsruhe (Germany) and Hursley (U.K.), for which we did not collect data. More details about the site structure and software are available from [147].

Requests for data from any client arrive at the Web site after being routed through the appropriate Internet NAP. These requests are passed to a load-balancing connection router that distributes them across several server nodes [3, 147]. These nodes retrieve the appropriate Web objects and transmit them across an internal ATM network and through the Internet to the clients.

7.2.2 Methodology

Stemm and Seshan monitored all the traffic coming into the site and obtained packet-level traces of this traffic by running the `tcpdump` [94] utility on a machine placed on the token ring connecting the load-balancing router to the server nodes (Figure 7.1). This machine was an IBM 150 Mhz Pentium Pro PC running BSD/OS 2.1 from BSDI [23]. They extracted and stored the first 350 bytes of every packet destined to TCP port 80 (the HTTP port), compressing this data on-the-fly using the `gzip` utility. In addition, the server also transmitted other data (such as streaming audio) on other ports. They did not capture this data due to limited storage space of the setup and the high data packet capture rates. Even while restricting packet capture to traffic on port 80, they collected approximately 1 GB of compressed packet headers per hour during peak periods. Periodically, the packet trace files were dumped onto tape. While dumping the trace files to tape, however, there were occasional periods during which the traffic was not monitored.

Due to the location of the trace collection machine and the asymmetric path of request and response data, only packets coming from the clients *into* the Web server complex were captured. These packets include the initial TCP SYN packet from the client, the HTTP request packets, as well as all TCP ACKs for data sent from the server. In particular, this implies that no data packets from the server to the client were captured. However, this does not prevent us from determining the packet sequence trace as a function of time, since we can estimate this from the sequence of TCP ACKs. In addition, we modified the TCP stack on a subset of the servers. These modified server nodes transmitted the TCP/IP header, together with information on the current sender congestion window size, smoothed round-trip time estimate, and number of duplicate ACKs causing the retransmission of all retransmitted packets on the token ring interface to the trace collection machine. This retransmission information combined with the ACK traces and the knowledge of the sender's TCP algorithms at the server allow us to reconstruct the outgoing data stream with high accuracy.

Trace Statistic	**Value**
Packets captured	1,540,312,422
Packets dropped	7,677,854
Average packet drop percentage	0.498%
Distinct client addresses	721,417
Total Bytes Collected	~189 GB

Table 7.2 Raw trace statistics.

Table 7.2 describes the raw statistics about the trace data. More details about these traces and a study of the stability of wide-area network performance based on them can be found in [17, 15].

7.3 TCP Emulation Engine

We post-processed the packet traces to generate a one-record summary of the progress of each TCP connection, including the times and sequence numbers of ACKs as well as the server-side retransmission notifications. We then organized the connection summaries into a database that allowed us to cluster connections from a single client and compare the TCP performance of different clients at different times.

To faithfully reproduce the state of the sender-side TCP implementation, we wrote a *TCP emulation engine* that took the captured ACKs as input and reproduced the evolution of the sender-side state variables (e.g., congestion window, round-trip time estimate, maximum packet sequence transmitted, etc.). The captured ACKs are sufficient for this because the change in any sender-side TCP state variable is triggered by the reception of an ACK or the expiration of a timer.

In essence, the engine is an implementation of a subset of the functions of a TCP sender at user-level, driven by a sequence of ACK traces. If the evolution of the sender's state were completely driven by the arrival of ACKs (as is usually the case if a connection experiences no losses), it is possible to reconstruct it with complete accuracy. We handle the complications due to timer events by using a number of heuristics, in addition to estimating the round-trip time by faithfully following the implementation on the Web server's TCP stack.

We validated our engine by comparing the times at which it predicted a retransmission event with the actual times that retransmissions occurred, as reported by the modified server nodes to the trace collector. When completed, the engine predicted the number of coarse timeouts, fast retransmissions, and slow-start retransmissions to within 0.75% of the correct number (even assuming that no retransmission notifications were lost by the packet capture process). As a result, all the window size calculations are also accurate to within this bound plus the underlying loss rate in the packet capture process (about 0.5% on average).

In summary, the TCP emulation engine is a tool that emulates the evolution of a TCP sender's state with a high degree of accuracy. The emulation engine, combined with the collected traces, allow us to analyze how TCP connections as used by Web browsers behave in today's Internet.

7.4 Analysis Results

In this section, we present the results from our analysis of the real-world behavior of TCP connections. We focus on the behavior of individual connections, analyzing how well the different TCP loss recovery mechanisms work in practice and looking for the presence of receiver bottlenecks.

7.4.1 Retransmissions and Loss Recovery

We classify the retransmissions of lost segments in TCP Reno into three categories—*fast retransmissions*, *timeouts* (Section 2.2.2) and *slow-start retransmissions*, which are retransmissions performed by the sender immediately after a timeout for subsequent packets that were presumed lost in the window. There are two situations that lead to coarse timeouts. In TCP Reno, the loss of multiple segments in a window usually leads to a coarse timeout. Coarse timeouts also occur when the number of duplicate ACKs is insufficient to trigger a fast retransmission.

Running the TCP emulation engine on the observed ACK trace showed that timeouts and the subsequent slow-start retransmissions are the predominant mechanism for loss recovery in TCP Reno. Specifically, over a 3 hour trace involving 1,650,103 connections and 285,979 individual retransmission events, we found that 49.3% of all retransmissions were due to timeouts, 43.8% were the result of a TCP fast retransmission, and 6.9% were the result of slow start retransmissions. That is, 56.2% of all retransmissions occurred soon after a coarse timeout.

We also characterized the state of the sender at the time a loss occurred into slow-start periods and congestion avoidance periods, to determine if either mode shows inherently different behavior from the other. Both congestion avoidance and slow-start have similar frequencies of loss: 82,181 (7%) out of 1,159,588 packets were lost in congestion avoidance and 354,566 (5%) out of 6,662,050 packets were lost in slow start.

From this analysis, we see that existing loss recovery techniques are not effective in recovering from packet losses in a timely manner and that new techniques must be developed to handle them.

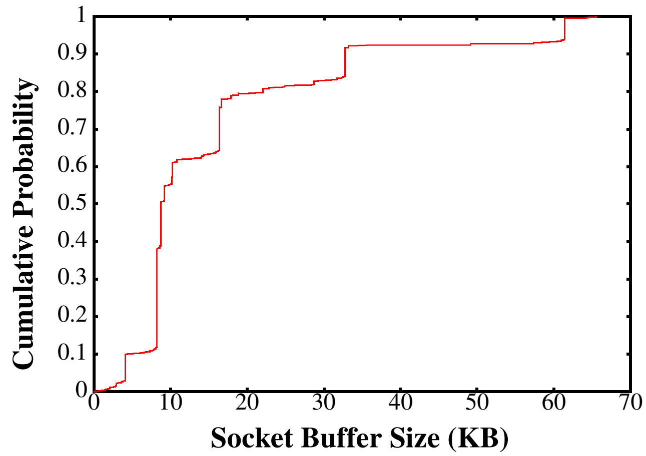

Figure 7.2 Cumulative distribution function of maximum receiver-advertised window sizes.

This result is disappointing because it implies that in practice, the more sophisticated data-driven loss recovery mechanisms are not sufficiently used and that there is a heavy dependence on timeouts for loss recovery. In Section 7.5, we discuss the effectiveness of TCP SACK in reducing the number of timeouts and describe an Enhanced Recovery (ER) technique to significantly improve performance.

7.4.2 Receiver-advertised window

Assuming that the source always has data to send, the amount of data transmitted by TCP at any time is primarily governed by the minimum of two parameters—the sender's congestion window, which tries to track the available bandwidth in the network, and the receiver's advertised window, which handles flow control. The maximum possible value for the latter parameter is equal to the socket buffer size chosen by the application when the connection is established. An excessively small value of this parameter could result in sub-optimal end-to-end performance if the receiver's advertised window is consistently smaller than the sender's congestion window, because the connection's bottleneck links could be under-utilized.

Figure 7.2 shows the cumulative distribution function of receiver socket buffer sizes for transfers to 32,000 hosts (over 1,650,103 connections) during a 3-hour period. The CDF shows distinct upswings at window sizes of 4 KB, 8 KB, 16 KB, etc.—values commonly used by receivers.

For each connection, we compared the receiver advertised window with the congestion window size as computed by the TCP engine. We found that in approximately 14% (233,906) of all connections the latter grew to be larger than the former multiple times during the connection; i.e., in 14% of all connections in this trace, the receiver advertised window limited the amount of data the TCP sender could have had outstanding. In these cases, the Web client could potentially have obtained a higher throughput had it employed a larger socket buffer and consequently have advertised a larger window.

From this analysis, we make the following recommendation*: future transport implementations should increase their default socket buffer size to avoid the receiver window from becoming a performance bottleneck.* In particular, default values of 4 KB used by some implementations are often too small.

7.5 Reducing the Occurrence of Timeouts

As discussed in Section 7.4.1, over 55% of all retransmissions in our connection trace of the Olympics Web server happened after one or more coarse timeouts. These timeouts keep the connection idle for periods from hundreds of milliseconds to seconds, stalling the steady flow of data to the client. Our analysis showed two main reasons for the occurrence of these timeouts:

1. *Timeout following a fast retransmission:* This happens when the TCP Reno sender is unable to recover from multiple losses within the same transmission window. Typically, the first loss is recovered by the fast retransmission mechanism, which is invoked after three duplicate ACKs reach the sender, but the other losses lead to timeouts.

2. *Insufficient duplicate ACKs:* Available bandwidths on many of today's Internet paths are low and many wireless links are characterized by inherently low channel bandwidths. Because transmission window sizes depend on the connection's bandwidth-delay product, low bandwidth paths keep windows small. When even a small number of losses occur in this small window, the number of arriving duplicate ACKs is not sufficient to trigger a fast retransmission. The result is a timeout-driven retransmission that keeps the link idle for long periods of time. This situation could also occur when most packets in a larger window are lost in the network.

Our solution to these problems has two components: Selective Acknowledgments (SACK) and Enhanced Recovery (ER). SACKs improve loss recovery and reduce the occurrence of timeouts when multiple losses occur in a large transmission window, while ER improves performance when windows are small—the common case on many Internet and wireless paths. In addition, these two techniques are orthogonal to each other and can coexist for better overall end-to-end performance when losses occur.

7.5.1 Selective Acknowledgments (SACK)

In Chapter 2 (Section 2.3.1), we discussed the IETF SACK mechanism based on RFC 2018 [92]. Recall that this document standardizes only the receiver behavior and SACK header format, leaving the details of the use of SACK information to the sender. The standard allows for up to three out-of-order, maximally contiguous blocks of data to be communicated to the sender, in "most-recent-first" order. The sender behavior is unspecified, except that it should invoke congestion control in response to congestion losses that the arrival of SACKs usually signify.

We now discuss some important details and tasks that the TCP sender performs in our SACK protocol. These mechanisms lead to better congestion control and also improve wireless performance.

1. *Accurate window management:* When duplicate ACKs signifying packet loss arrive in TCP Reno, the sender increments its congestion window by one maximum-sized segment for each incoming duplicate ACK. In general, this is inaccurate because it is not tied to exactly how many data bytes have left the network. For instance, if some duplicate ACKs get lost, the increase is pessimistic; on the other hand, if the sender previously transmitted variable-sized (especially small) segments, the corresponding increase is optimistic. With SACKs, it is possible to avoid this and increment the congestion window precisely by the number of bytes that have reached. This information is available in the SACK blocks, and is relatively robust to ACK loss because information about each block is communicated in at least three different SACKs from the receiver. This is the same approach used in Mathis and Mahdavi's FACK scheme [91].

2. *Fast recovery modification:* In TCP Reno, all duplicate ACKs beyond the third one cause the sender to increment its congestion window by one segment. When more than half the window has been successfully acknowledged, the sender enters the fast recovery phase, transmitting a new segment for each arriving duplicate ACK (Section 2.2.2).

   While this approach does reduce the effective window during fast recovery to half its value at the time of loss, the *rate* at which new segments are transmitted is still approximately equal to the old rate. This is because a new segment is transmitted for each incoming duplicate ACK—

and the rate at which ACKs arrive is approximately equal to the rate at which segments were originally transmitted. If a router en route was incapable of sustaining the original data rate, then it is unlikely to sustain the same rate during fast recovery. This could lead to further losses in this period and require coarse-grained timeouts for recovery.

In contrast, our solution (also independently suggested by Hoe [55]), is to ensure that the transmission rate of new segments is halved during fast recovery. Rather than wait for half the window and then clock data out at the original rate, we transmit one segment for every other incoming ACK. This ensures that the transmission rate is halved, in accordance with congestion control principles. Once fast recovery is complete, the sender's congestion window is set to half the original value at the time of loss.

3. *Integration with Explicit Loss Notification (ELN):* In Chapter 5, we described the details of ELN and its benefits in distinguishing congestion from wireless corruption. By combining ELN messages with SACK information, efficient recovery from wireless losses is possible, especially over wireless networks with large bandwidth-delay products. To understand the benefits of this, consider the following example. Suppose that packets 1, 3 and 5 have been lost due to wireless corruption, all within the same transmission window. With only cumulative ACKs and ELN, each ELN indication triggered by duplicate ACKs indicates the corruption loss of the earliest packet, 1. Thus, it is not clear to the sender why packets 3 and 5 were lost. With SACK, it is possible to indicate exactly which SACK holes correspond to corruption losses, and for the sender to appropriately react to this information.

4. *Handling variable-sized segments:* In standard TCP without SACK, a retransmission usually sends out one MSS worth of data into the network. While this is appropriate for transfers where the sender originally segmented data into MSS-sized packets, it often leads to redundant retransmissions of data already at the receiver. Such variable-sized segments arise in several applications such as interactive terminal sessions (e.g., `telnet`); over low-bandwidth, high-latency wireless networks, these redundant retransmissions excessively reduce goodput and degrade interactive performance.

   It is possible, but not easy or elegant, to solve this problem without the use of SACKs, by maintaining additional state in the TCP sender's control block. However, SACKs provide an elegant solution to this problem, since the sender is now aware of exactly which bytes beyond the miss-

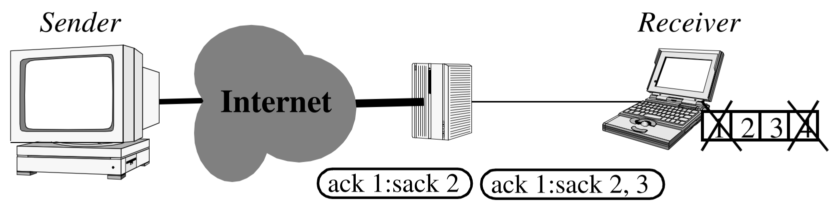

Figure 7.3 Illustration of how small windows affect data-driven loss recovery. Only two duplicate ACKs with SACK are generated, which is insufficient information for the sender to deduce that segment 1 is missing and not just reordered in the network.

ing segment are already at the receiver. Hence, rather than retransmit an MSS worth of data beginning at the first missing one, the sender only retransmits data up to the first selectively acknowledged byte beyond the cumulative ACK.

The performance benefits of SACK are significant for connections whose bandwidth-delay product is large, where multiple losses are likely to occur within a window [42]. Examples of such networks include satellite networks [53] and high-bandwidth terrestrial networks [109].

7.5.2 Enhanced Recovery (ER)

The use of TCP SACK has often been suggested as a technique to improve loss recovery and avoid timeouts unless there is persistent congestion in the network [42]. While this is true in networks with large bandwidth-delay products, the benefits of SACK in preventing timeouts over current (late 1990s) Internet paths and many wireless networks is questionable. If transmission windows are small, then even a small number of losses in the window are not recoverable without timing out, even if the SACK protocol is deployed. An example of such a situation is shown in Figure 7.3.

The problem arises because the sender needs to ensure that a segment is in fact lost and not just *reordered* in the network before initiating a retransmission. Unfortunately, packet reordering is an increasingly common phenomenon on many Internet paths [110], as well as in packet radio networks (e.g., Metricom's Ricochet network [77]). The TCP Reno sender waits for three duplicate ACKs before retransmitting the missing segment; this is a heuristic based on observations of how much reordering typical paths display. When windows are small, not enough duplicate ACKs arrive for the sender to make this distinction, necessitating a timeout.

Observe that in these situations, the information provided by SACKs is not helpful in deducing that reordering has not occurred and in avoiding timeouts (Figure 7.3). A detailed analysis of the trace data based on the output of the TCP emulation engine (Section 7.3) showed that out of the 422,025 coarse timeouts over a 3-hour period, SACKs could have avoided at most 18,713 (4.43%) of them. That is, situations where windows are large enough to have prevented unnecessary timeouts using SACKs are rather infrequent. It is therefore clear that an alternative technique is needed to recover from the bulk of losses, without incurring expensive timeouts.

Our solution to this problem uses the following observations and design principles:

1. *Entire windows are often not lost:* During the same 3-hour period, approximately 403,312 (95.6%) coarse timeouts occurred as a result of insufficient ACK information (i.e., an insufficient number of duplicate ACKs arrived to trigger a retransmission). Of these timeouts, no duplicate ACKs arrived at all for 70% of them. In these situations the network is most likely experiencing severe congestion, since the entire window (or all ACKs) are lost. The best solution for the sender to wait for a coarse timeout to occur before transmitting any packets. However, the remaining 30% of these timeouts occurred even though the entire window was *not* lost, and the fast retransmission mechanism was insufficient. Only 4.43% of these had windows large enough that the deployment of SACKs would have avoided the resulting timeout.

2. *Conservation of packets:* This is the guiding principle used in TCP's transmission algorithms. When an ACK (new or duplicate) arrives at the sender, the sender is assured that at least one segment has left the data pipe and is in the receiver's buffers. Therefore, it can transmit at least as many bytes as have left the pipe without significantly affecting the load it imposes on the network.

We make the following enhancements to the sender algorithm to greatly reduce the occurrence of timeouts. These apply when the window is small, less than six segments.

When the first duplicate ACK arrives, we know that one segment has cleared the pipe and is at the receiver. It is likely that the segment indicated by the duplicate ACK is lost, but the sender cannot be sure of this because of the possibility of packet reordering. Following the conservation of packets principle, the sender can now transmit one segment to the receiver. Because the retransmission of a segment could prove redundant, the sender transmits a *new* segment from beyond the "right edge" of the current window. Note that this does not violate any congestion control principles because of packet conservation.

Now, if this new segment reaches the receiver, it will generate another duplicate ACK. As long as the number of duplicate ACKs is smaller than three (the threshold for fast retransmissions), the sender transmits new segments in this fashion, probing the network for sustained congestion or loss. Upon the arrival of the third duplicate ACK, the sender performs a retransmission as usual. We call this form of loss recovery using these new "probe" segments *enhanced recovery* (ER). Note that we have not violated standard congestion control procedures by doing this—we transmit a segment only when one has left the data pipe. In addition, if SACK information were also available, the arrival of an ACK with a SACK block indicating the reception of one of the newly transmitted segments is a strong indicator that the original segment was lost, independent of whether three duplicate ACKs arrive or not. Thus, this mechanism will improve performance even when the sender's window is small and losses occur and prevent the detrimental effects of long timeouts unless there is *sustained* network congestion.

In practice, we deploy ER together with Hoe's Newreno techniques, which are orthogonal to ER and complement ER well. Newreno, which keeps the sender in fast recovery if the new ACK that arrives after a fast retransmission is "partial," usually recovers from multiple losses in a large transmission window without incurring a timeout. In this respect, it achieves the same results as SACK, without any changes required at the receiver. However, the rate at which lost segments in the window are recovered is roughly one segment per round trip time, rather than multiple segments in each round-trip duration. ER and Newreno complement each other well because the former prevents unnecessary timeouts when windows are small, while the latter prevents timeouts when windows are large.

We now show that timeouts occur in our protocol only when congestion is persistent. There are three times when timeouts occur. First, when the entire transmission window, or all ACKs for a window, are lost in the network. Because the sender now has no ACK information to perform data-driven loss recovery, a timeout occurs. Second, when a retransmission of a lost segment is also lost. This is also a sign of sustained congestion since the loss situation has persisted for on the order of a round-trip time. Finally, with ER, timeouts may also occur if a newly sent "probe" segment from the right edge of the window was lost, or if the sender did not receive a duplicate ACK for it. As before, this is also a likely sign of sustained congestion in the network. We note here that if any of these duplicate ACKs had ELN information on them (as explained in Chapter 4), then the corresponding losses occurred because of corruption; these trigger retransmissions *without* the associated congestion control actions at the sender.

7.6 Implementation

In this section, we discuss some details of our implementations of SACK and ER. Our implementation is in the BSD/OS operating system from BSDI [23]. The SACK implementation is conformant with RFC 2018.

7.6.1 SACK

SACK involves changes to both the sender and receiver TCP implementations. RFC 2018 describes the receiver mechanism in detail, leaving large parts of the sender algorithms to specific implementations. Thus, we will focus mostly on the sender details of our implementation and the points discussed in Section 7.5.1.

At the receiver, out-of-sequence segments cause entries to be updated (or newly added) to a list of blocks. Blocks are maximal contiguous regions of received data in the transmission sequence beyond the current cumulative ACK. Whenever a gap is detected in the transmission sequence, duplicate cumulative ACKs with up to three SACK blocks are sent to the TCP sender. The three chosen blocks are ones with most recent activity in them, which is a heuristic that is robust to the loss of some ACKs on the reverse path. These blocks are marshalled into the TCP header in a special SACK option.

At the sender, we maintain a list of holes, which correspond to contiguous gaps in the receiver's data sequence. The arrival of a SACK causes this list to be updated because each SACK usually contains information about the receipt of some data item, information about which was previously unknown to the sender. Each SACK arrival also results in a check to decide how many items in the list of holes are now eligible for retransmission. To distinguish between reordered segments and lost ones, a hole is marked eligible if we know from SACK information that a segment greater than or equal to at least three segments from the missing one has been received successfully.

A hole marked eligible for retransmission is not necessarily retransmitted the moment it is marked. Following the principle of packet conservation, we keep track of how many bytes have currently been acknowledged by successively arriving SACKs. Every time this number increases by twice the MSS, up to one MSS-worth of data may be transmitted. At this point, the sender traverses its list of holes in order, picking out one MSS-worth of data to be retransmitted. If there are no holes eligible for retransmission, *and* if a segment can be transmitted, then a new one is sent out. That is, for a bulk transfer, every other duplicate ACK with SACK information triggers the transmission of a segment, with lost segments (after accounting for possible reordering) sent implicitly at higher priority than new ones.

There are some marked differences between this algorithm and the one TCP Reno normally uses during fast retransmission and recovery. First, in TCP Reno, a retransmission automatically causes an MSS-worth of data starting from the cumulative ACK to be sent, whereas with SACK we retransmit one or more holes reducing the chances that no previously received bytes are resent. Second, in TCP Reno (and Newreno), the duplicate ACKs after the third one cause new segments to be transmitted, not older ones, because the sender does not know for sure which specific segments in the original window are missing. With SACK, typically every other duplicate ACK causes the transmission of an eligible lost segment; only if no segments are eligible is a new segment transmitted. Finally, with SACK, more precise congestion window management is possible because the number of bytes acknowledged by each duplicate ACK is more exact than the rough estimate of one MSS that duplicate ACKs alone indicate. This is especially beneficial in situations when the ACK path is loss-prone or when the original transmissions were not in multiples of the connection's MSS.

7.6.2 ER

One important advantage of the ER scheme is that it involves changes only to the TCP sender and not to the receiver, unlike SACK. This makes it easier to incrementally deploy ER in the Internet and reap its benefits.

If a duplicate ACK arrives when the sender's congestion window is less than or equal to six segments, it transmits one new segment. At this time, it also increments a variable in the control block, `er_sent`, which keeps track of the number of new segments transmitted in each recovery phase.

When the third duplicate ACK arrives and a fast retransmission occurs, the congestion window is decremented by the number of bytes sent in the ER phase. This is because the TCP Reno sender sets its congestion window after a fast retransmission as:

```
tp→snd_cwnd += tp→t_dupacks * tp→t_maxseg;
```

Because ER would have already sent out `tp→er_sent*tp→t_maxseg` bytes, we now adjust the congestion window as:

```
tp→snd_cwnd -= tp→er_sent * tp→t_maxseg;
```

If the sender and receiver have negotiated the SACK option, then ER is terminated upon the arrival of a SACK block acknowledging the receipt of the newly sent segment, even if three duplicate cumulative ACKs have not arrived. This is sufficient to guard against possible reordering in the network.

Finally, if the connection itself is short, there may not be any new segments to transmit in the ER phase when new segments are transmitted. There are two ways of handling this situation:

1. We could transmit only the header of a zero-byte segment corresponding to the last byte of the connection. This approach does not add significantly to the state of congestion if the unit of buffering in the routers is in terms of bytes, as opposed to packets. Then, upon three duplicate ACKs, a fast retransmission of the missing segment is possible as usual.
2. We could lower the fast retransmission threshold when the window is small *and* there are no new segments to transmit. Then if no new segment can be sent upon the arrival of the first (or second) duplicate ACK, the missing one is retransmitted. This enables quicker loss recovery than (1) above and does not introduce any unnecessary probe packets into the network. We adopt this approach in our implementation.

7.7 Performance

In this section, we present measurements of our SACK implementation and simulation and implementation results of ER.

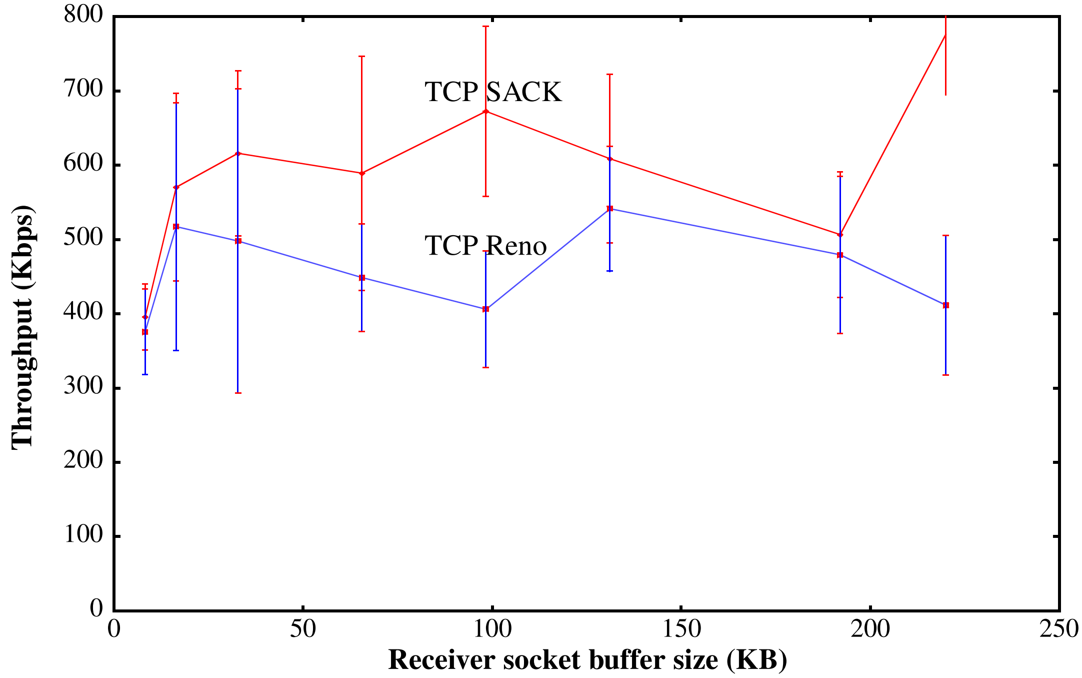

Figure 7.4 Measured throughputs for TCP Reno and SACK over a 16-hop Internet path as a function of the receiver socket buffer size. The performance improvement due to SACK on his particular path is between 5 and 30% compared to Reno. The vertical errorbars show the standard deviations of the measurements.

7.7.1 SACK

We measured the performance of our SACK implementation on a connection over a randomly chosen Internet path to verify the correctness of the implementation and to understand what the typical degree of performance improvement might be when it is deployed in the Internet. One of the hosts was at UC Berkeley (CA) and the other was at IBM Watson Research Center (NY), with 16 Internet hops between them. We performed these experiments during the day when congestion on the Internet caused losses. Each experiment consisted of a 2 MB TCP transfer, repeated ten times at each receiver socket buffer size. The round-trip time between the hosts varied between 90 and 200 ms with an average of 135 ms during the experiments; the maximum bandwidth between the two end hosts was about 1.35 Mbps.

The results of the experiments are shown in Figure 7.4, which plots the TCP throughput as a function of the socket buffer size for SACK and TCP Reno. We find that when socket buffers are small, there is little difference between the two protocols. This is because at small socket buffer sizes,

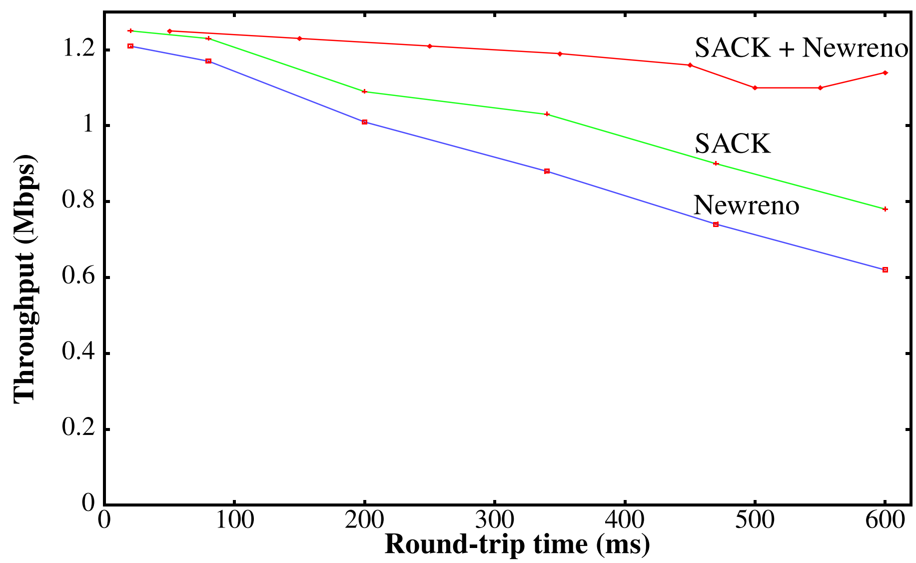

Figure 7.5 Performance of SACK+Newreno, SACK and Newreno as a function of connection round-trip time. (Measurements taken by Tom Henderson, U.C. Berkeley).

TCP transmission windows are small because the buffer size bounds the receiver-advertised window, which limits the number of segments in transit at any point in time. As the socket buffer size increases, the performance of both protocols improves markedly until the buffer size reaches the connection's bandwidth-delay product of 23 KB. At buffer sizes larger than this, SACK performance is between 15% and 30% better than Reno because of its ability to recover from multiple losses in a transmission window without incurring as many timeouts. Thus, over Internet paths where the bandwidth-delay products are large enough to permit large transmission windows, SACK performance is perceptibly better than Reno when losses occur.

Figure 7.5 shows the performance of SACK+Newreno, SACK and Newreno as a function of the connection round-trip time for long TCP transfers with a socket buffer size of 240 KB. No other traffic was present during these experiments, so all losses were caused due to overflows induced by TCP's probing for extra bandwidth.

We find that even when there is no competing traffic, the occurrence of multiple losses in a window significantly degrades the performance of Newreno as round-trip times increase. SACK does better because it recovers more quickly, but the each loss in the window reduces the window by a factor of two. The sender does recover from all losses without a timeout, but its window drops to a small value after this. Then, it takes several round-trips to build again to the bandwidth-delay prod-

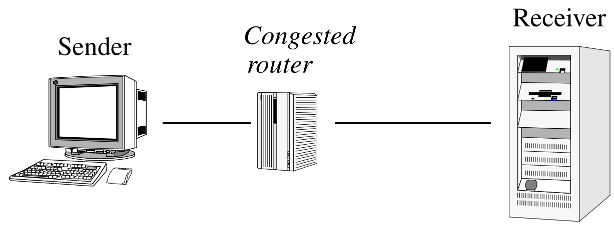

Figure 7.6 Topology for ER simulation experiments.

uct, which explains why this implementation of SACK shows performance degradation at large round-trip times. Finally, the performance of SACK+Newreno is close to optimal, independent of the round-trip time of the connection. This is because SACK promptly recovers from multiple losses and Newreno prevents the window from retracting multiple times. We also note that we do not need Newreno with SACK to perform well; the same benefit can be obtained by ensuring that TCP does not retract its window multiple times in the same round-trip epoch.

7.7.2 ER

We performed an *ns* simulation using the simple topology shown in Figure 7.6 to validate and test the ER algorithm. We generated cross-traffic using a combination of UDP (constant rate) and TCP (Web-like) workloads to force the measured TCP sender to cope with frequent losses at the bottleneck router. These frequent losses resulted in a small average congestion window, affecting data-driven loss recovery and forcing numerous timeouts. This experiment was done to recreate situations that occur in the Olympic Web traces in a controlled manner, where congestion is the result of a large number of flows competing for scarce resources at network bottlenecks. Each simulation experiment consisted of a single TCP transfer from the sender to receiver for a duration of 10 seconds, using SACK+Newreno and ER+Newreno at the sender.

Figure 7.7 shows the sequence plots for TCP with SACK+Newreno and ER+Newreno. As indicated by the large gaps in the sequence plot, the SACK transfer experiences many more coarse timeouts than the transfer using ER. These coarse timeouts are a result of frequent losses that generate two or fewer duplicate ACKs. These duplicates are not sufficient to trigger fast retransmissions at the sender. Therefore, in a traditional TCP implementation the loss is only recovered from after a coarse timeout occurs. With ER, the sender transmits a new packet when each these duplicate ACKs arrive, which elicit additional duplicate ACKs from the receiver. These additional duplicates

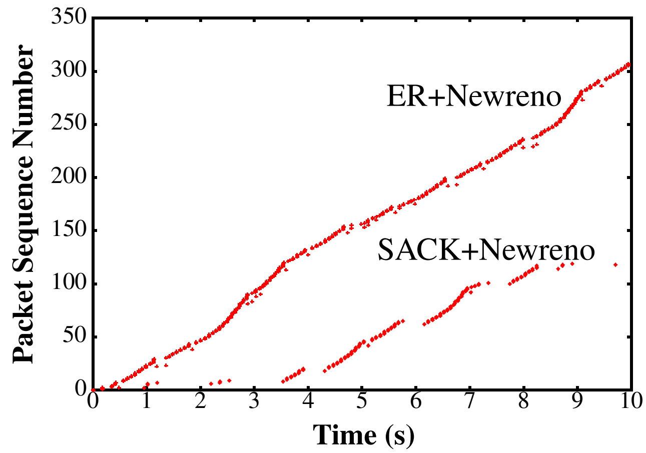

Figure 7.7 TCP with and without ER.

allow the sender to use fast retransmissions to recover from the losses. By eliminating the coarse timeouts, the sender using ER completes the transfer more than twice as fast as the SACK transmitter.

We performed the same experiments using our implementations of SACK+Newreno and ER+Newreno over busy cross-country Internet and wireless paths. We used a buffer size of 32 KB at the receiver. A sample result from one of the runs over the same path as the experiments in Section 7.7.1 is shown in Figure 7.8, which depicts a 5.5-second sequence number trace of the two versions of TCP. We find that SACK+Newreno incurs two long timeouts in this trace, while ER+Newreno successfully prevents any timeouts, leading to a performance improvement of over 100%. We have observed similar results and corresponding performance improvements for ER in experiments conducted over other congested and loss-prone network paths.

7.8 Summary

In this chapter, we analyzed the problems that arise due to small TCP transmission windows and solved them by enhancing TCP's data-driven loss recovery mechanism. Small transmission windows occur because of the inherently low channel bandwidths in many wireless networks and the short lengths of today's Web transfers. The result of small TCP windows is that data-driven loss recovery using fast retransmissions is not triggered often: most losses require expensive timeouts to recover from, keeping the connection idle for long periods of time.

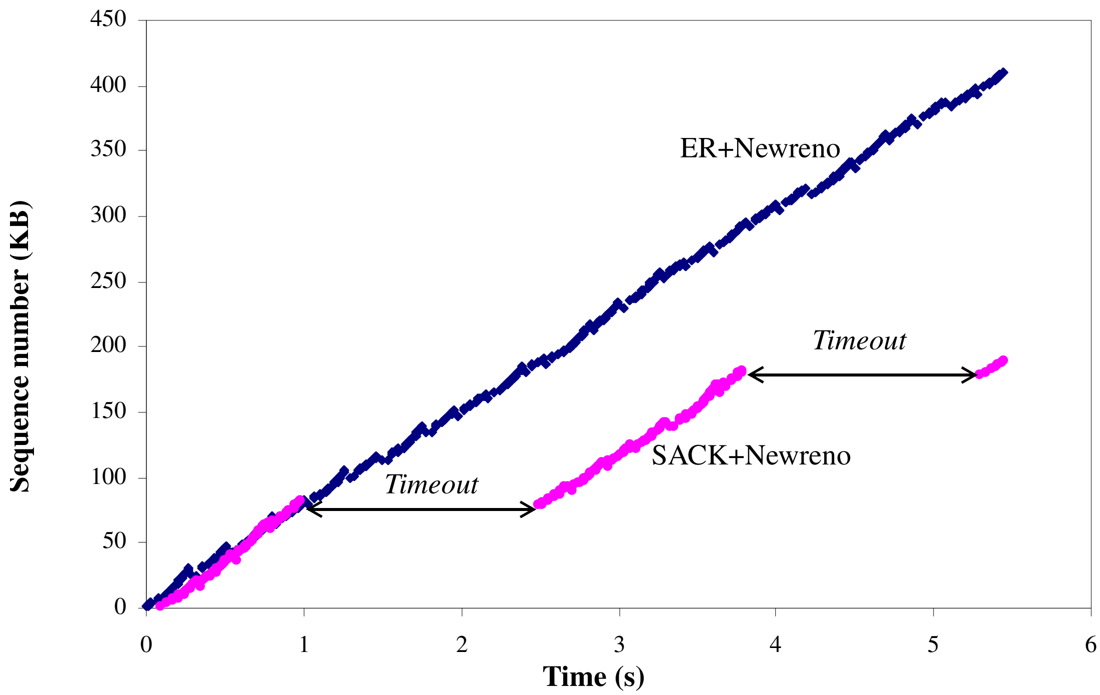

Figure 7.8 Sequence number traces of ER and SACK for a short wide-area transfer. ER successfully prevents sender timeouts, whereas SACK incurs two long timeouts that lead to less than half the throughput of ER in this experiment.

We described two enhancements to TCP's loss recovery algorithms to reduce the occurrence of expensive timeouts, relying instead on data-driven modes of loss recovery: SACK and ER. After discussing the benefits and limitations of SACK, we present a new Enhanced Recovery (ER) scheme that uses the principle of packet conservation to probe the network's state of congestion upon the arrival of early duplicate ACKs in a small window. The successful receipt of these probe segments leads to additional duplicate ACKs from the receiver, which the sender uses to distinguish reordered segments from genuine losses and therefore retransmit missing segments. Together with Newreno, ER ensures that TCP timeouts now occur only if losses are persistent in the network.