Simulation-based Comparisons of Tahoe, Reno, and SACK TCP

Kevin Fall and Sally Floyd*

Lawrence Berkeley National Laboratory
One Cyclotron Road, Berkeley, CA 94720
kfall@ee.lbl.gov, floyd@ee.lbl.gov

Abstract

This paper uses simulations to explore the benefits of adding selective acknowledgments (SACK) and selective repeat to TCP. We compare Tahoe and Reno TCP, the two most common reference implementations for TCP, with two modified versions of Reno TCP. The first version is New-Reno TCP, a modified version of TCP without SACK that avoids some of Reno TCP's performance problems when multiple packets are dropped from a window of data. The second version is SACK TCP, a conservative extension of Reno TCP modified to use the SACK option being proposed in the Internet Engineering Task Force (IETF). We describe the congestion control algorithms in our simulated implementation of SACK TCP and show that while selective acknowledgments are not required to solve Reno TCP's performance problems when multiple packets are dropped, the absence of selective acknowledgments does impose limits to TCP's ultimate performance. In particular, we show that without selective acknowledgments, TCP implementations are constrained to either retransmit at most one dropped packet per round-trip time, or to retransmit packets that might have already been successfully delivered.

1 Introduction

In this paper we illustrate some of the benefits of adding selective acknowledgment (SACK) to TCP. Current implementations of TCP use an acknowledgment number field that contains a cumulative acknowledgment, indicating the TCP receiver has received all of the data up to the indicated byte. A selective acknowledgment option allows receivers to additionally report non-sequential data they have received. When coupled with a selective retransmission policy implemented in TCP senders, considerable savings can be achieved.

Several transport protocols have provided for selective acknowledgment (SACK) of received data. These include NETBLT [CLZ87], XTP [SDW92], RDP [HSV84] and VMTP [Che88]. The first proposals for adding SACK to TCP [BJ88, BJZ90] were later removed from the TCP RFCs (Request For Comments) [BBJ92] pending further research. The current proposal for adding SACK to TCP is given in [MMFR96]. We use simulations to show how the SACK option defined in [MMFR96] can be of substantial benefit relative to TCP without SACK.

The simulations are designed to highlight performance differences between TCP with and without SACK. In this paper, Tahoe TCP refers to TCP with the Slow-Start, Congestion Avoidance, and Fast Retransmit algorithms first implemented in 4.3 BSD Tahoe TCP in 1988. Reno TCP refers to TCP with the earlier algorithms plus Fast Recovery, first implemented in 4.3 BSD Reno TCP in 1990.

Without SACK, Reno TCP has performance problems when multiple packets are dropped from one window of data. These problems result from the need to await a retransmission timer expiration before reinitiating data flow. Situations in which this problem occurs are illustrated later in this paper (for example, see Section 6.4).

Not all of Reno's performance problems are a necessary consequence of the absence of SACK. To show why, we implemented a variant of the Reno algorithms in our simulator, called New-Reno. Using a suggestion from Janey Hoe [Hoe95, Hoe96], New-Reno avoids many of the retransmit timeouts of Reno without requiring SACK. Nevertheless, New-Reno does not perform as well as TCP with SACK when a large number of packets are dropped from a window of data. The purpose of our discussion of New-Reno is to clarify the fundamental limitations of the absence of SACK.

In the absence of SACK, both Reno and New-Reno senders can retransmit at most one dropped packet per round-trip time, even if senders recover from multiple

*This work was supported by the Director, Office of Energy Research, Scientific Computing Staff, of the U.S. Department of Energy under Contract No. DE-AC03-76SF00098.

drops in a window of data without waiting for a retransmit timeout. This characteristic is not shared by Tahoe TCP, which is not limited to retransmitting at most one dropped packet per round-trip time. However, *it is a fundamental consequence of the absence of SACK that the sender has to choose between the following strategies to recover from lost data:*

1. retransmitting at most one dropped packet per round-trip time, or
2. retransmitting packets that might have already been successfully delivered.

Reno and New-Reno use the first strategy, and Tahoe uses the second.

To illustrate the advantages of TCP with SACK, we show simulations with SACK TCP, using the SACK implementation in our simulator. SACK TCP is based on a conservative extension of the Reno congestion control algorithms with the addition of selective acknowledgments and selective retransmission. With SACK, a sender has a better idea of exactly which packets have been successfully delivered as compared with comparable protocols lacking SACK. Given such information, a sender can avoid unnecessary delays and retransmissions, resulting in improved throughput. We believe the addition of SACK to TCP is one of the most important changes that should be made to TCP at this time to improve its performance.

In Sections 2 through 5 we describe the congestion control and packet retransmission algorithms in Tahoe, Reno, New-Reno, and SACK TCP. Section 6 shows simulations with Tahoe, Reno, New-Reno, and SACK TCP in scenarios ranging from one to four packets dropped from a window of data. Section 7 shows a trace of Reno TCP taken from actual Internet traffic, showing that the performance problems of Reno without SACK are of more than theoretical interest. Finally, Section 8 discusses possible future directions for TCP with selective acknowledgments, and Section 9 gives conclusions.

2 Tahoe TCP

Modern TCP implementations contain a number of algorithms aimed at controlling network congestion while maintaining good user throughput. Early TCP implementations followed a go-back-n model using cumulative positive acknowledgment and requiring a retransmit timer expiration to re-send data lost during transport. These TCPs did little to minimize network congestion.

The Tahoe TCP implementation added a number of new algorithms and refinements to earlier implementations. The new algorithms include *Slow-Start*, *Congestion Avoidance*, and *Fast Retransmit* [Jac88]. The refinements include a modification to the round-trip time estimator used to set retransmission timeout values. All modifications have been described elsewhere [Jac88, Ste94].

The Fast Retransmit algorithm is of special interest in this paper because it is modified in subsequent versions of TCP. With Fast Retransmit, after receiving a small number of duplicate acknowledgments for the same TCP segment (*dup ACKs*), the data sender infers that a packet has been lost and retransmits the packet without waiting for a retransmission timer to expire, leading to higher channel utilization and connection throughput.

3 Reno TCP

The Reno TCP implementation retained the enhancements incorporated into Tahoe, but modified the Fast Retransmit operation to include *Fast Recovery* [Jac90]. The new algorithm prevents the communication path ("pipe") from going empty after Fast Retransmit, thereby avoiding the need to Slow-Start to re-fill it after a single packet loss. Fast Recovery operates by assuming each dup ACK received represents a single packet having left the pipe. Thus, during Fast Recovery the TCP sender is able to make intelligent estimates of the amount of outstanding data.

Fast Recovery is entered by a TCP sender after receiving an initial threshold of dup ACKs. This threshold, usually known as *tcprexmtthresh*, is generally set to three. Once the threshold of dup ACKs is received, the sender retransmits one packet and reduces its congestion window by one half. Instead of slow-starting, as is performed by a Tahoe TCP sender, the Reno sender uses additional incoming dup ACKs to clock subsequent outgoing packets.

In Reno, the sender's *usable* window becomes $min(awin, cwnd + ndup)$ where $awin$ is the receiver's advertised window, $cwnd$ is the sender's congestion window, and $ndup$ is maintained at 0 until the number of dup ACKs reaches *tcprexmtthresh*, and thereafter tracks the number of duplicate ACKs. Thus, during Fast Recovery the sender "inflates" its window by the number of dup ACKs it has received, according to the observation that each dup ACK indicates some packet has been removed from the network and is now cached at the receiver. After entering Fast Recovery and retransmitting a single packet, the sender effectively waits until half a window of dup ACKs have been received, and then sends a new packet for each additional dup ACK that is received. Upon receipt of an ACK for new data (called a "recovery ACK"), the sender exits Fast Recovery by setting $ndup$ to 0. Fast Recovery is illustrated in more detail in the simulations in Section 6.

Reno's Fast Recovery algorithm is optimized for the case when a single packet is dropped from a window of data. The Reno sender retransmits at most one dropped packet per round-trip time. Reno significantly improves upon the behavior of Tahoe TCP when a single packet is dropped from a window of data, but can suffer from performance problems when multiple packets are dropped from a window of data. This is illustrated in the simulations in Section 6 with three or more dropped packets. The problem is easily constructed in our simulator when a Reno TCP connection with a large congestion window suffers a burst of packet losses after slow-starting in a network with drop-tail gateways (or other gateways that fail to monitor the average queue size).

4 New-Reno TCP

We include New-Reno TCP in this paper to show how a simple change to TCP makes it possible to avoid some of the performance problems of Reno TCP without the addition of SACK. At the same time, we use New-Reno TCP to explore the fundamental limitations of TCP performance in the absence of SACK.

The New-Reno TCP in this paper includes a small change to the Reno algorithm at the sender that eliminates Reno's wait for a retransmit timer when multiple packets are lost from a window [Hoe95, CH95]. The change concerns the sender's behavior during Fast Recovery when a *partial ACK* is received that acknowledges some but not all of the packets that were outstanding at the start of that Fast Recovery period. In Reno, partial ACKs take TCP out of Fast Recovery by "deflating" the usable window back to the size of the congestion window. In New-Reno, partial ACKs do not take TCP out of Fast Recovery. Instead, partial ACKs received during Fast Recovery are treated as an indication that the packet immediately following the acknowledged packet in the sequence space has been lost, and should be retransmitted. Thus, when multiple packets are lost from a single window of data, New-Reno can recover without a retransmission timeout, retransmitting one lost packet per round-trip time until all of the lost packets from that window have been retransmitted. New-Reno remains in Fast Recovery until all of the data outstanding when Fast Recovery was initiated has been acknowledged.

The implementations of New-Reno and SACK TCP in our simulator also use a "maxburst" parameter. In our SACK TCP implementation, the "maxburst" parameter limits to four the number of packets that can be sent in response to a single incoming ACK, even if the sender's congestion window would allow more packets to be sent. In New-Reno, the "maxburst" parameter is set to four packets outside of Fast Recovery, and to two packets during Fast Recovery, to more closely reproduce the behavior of Reno TCP during Fast Recovery. The "maxburst" parameter is really only needed for the first window of packets that are sent after leaving Fast Recovery. If the sender had been prevented by the receiver's advertised window from sending packets during Fast Recovery, then, without "maxburst", it is possible for the sender to send a large burst of packets upon exiting Fast Recovery. This applies to Reno and New-Reno TCP, and to a lesser extent, to SACK TCP. In Tahoe TCP the Slow-Start algorithm prevents bursts after recovering from a packet loss. The bursts of packets upon exiting Fast Recovery with New-Reno TCP are illustrated in Section 6 in the simulations with three and four packet drops. Bursts of packets upon exiting Fast Recovery with Reno TCP are illustrated in [Flo95].

[Hoe95] recommends an additional change to TCP's Fast Recovery algorithms. She suggests the data sender send a new packet for every two dup ACKs received during Fast Recovery, to keep the "flywheel" of ACK and data packets going. This is not implemented in "New-Reno" because we wanted to consider the minimal set of changes to Reno needed to avoid unnecessary retransmit timeouts.

5 SACK TCP

The SACK TCP implementation in this paper, called "Sack1" in our simulator, is also discussed in [Flo96b, Flo96a].[1] The SACK option follows the format in [MMFR96]. From [MMFR96], the SACK option field contains a number of SACK blocks, where each SACK block reports a non-contiguous set of data that has been received and queued. The first block in a SACK option is required to report the data receiver's most recently received segment, and the additional SACK blocks repeat the most recently reported SACK blocks [MMFR96]. In these simulations each SACK option is assumed to have room for three SACK blocks. When the SACK option is used with the Timestamp option specified for TCP Extensions for High Performance [BBJ92], then the SACK option has room for only three SACK blocks [MMFR96]. If the SACK option were to be used with both the Timestamp option and with T/TCP (TCP Extensions for Transactions) [Bra94], the TCP option space would have room for only two SACK blocks.

[1] The 1990 "Sack" TCP implementation on our previous simulator is from Steven McCanne and Sally Floyd, and does not conform to the formats in [MMFR96]. The new "Sack1" implementation contains major contributions from Kevin Fall, Jamshid Mahdavi, and Matt Mathis.

The congestion control algorithms implemented in our SACK TCP are a conservative extension of Reno's congestion control, in that they use the same algorithms for increasing and decreasing the congestion window, and make minimal changes to the other congestion control algorithms. Adding SACK to TCP does not change the basic underlying congestion control algorithms. The SACK TCP implementation preserves the properties of Tahoe and Reno TCP of being robust in the presence of out-of-order packets, and uses retransmit timeouts as the recovery method of last resort. The main difference between the SACK TCP implementation and the Reno TCP implementation is in the behavior when multiple packets are dropped from one window of data.

As in Reno, the SACK TCP implementation enters Fast Recovery when the data sender receives *tcprexmtthresh* duplicate acknowledgments. The sender retransmits a packet and cuts the congestion window in half. During Fast Recovery, SACK maintains a variable called `pipe` that represents the estimated number of packets outstanding in the path. (This differs from the mechanisms in the Reno implementation.) The sender only sends new or retransmitted data when the estimated number of packets in the path is less than the congestion window. The variable `pipe` is incremented by one when the sender either sends a new packet or retransmits an old packet. It is decremented by one when the sender receives a dup ACK packet with a SACK option reporting that new data has been received at the receiver.[2]

Use of the `pipe` variable decouples the decision of *when* to send a packet from the decision of *which* packet to send. The sender maintains a data structure, the *scoreboard* (contributed by Jamshid Mahdavi and Matt Mathis), that remembers acknowledgments from previous SACK options. When the sender is allowed to send a packet, it retransmits the next packet from the list of packets inferred to be missing at the receiver. If there are no such packets and the receiver's advertised window is sufficiently large, the sender sends a new packet.

When a retransmitted packet is itself dropped, the SACK implementation detects the drop with a retransmit timeout, retransmitting the dropped packet and then slow-starting.

The sender exits Fast Recovery when a recovery acknowledgment is received acknowledging all data that was outstanding when Fast Recovery was entered.

The SACK sender has special handling for partial ACKs (ACKs received during Fast Recovery that advance the Acknowledgment Number field of the TCP header, but do not take the sender out of Fast Recovery). For partial ACKs, the sender decrements `pipe` by two packets rather than one, as follows. When Fast Retransmit is initiated, `pipe` is effectively decremented by one for the packet that was assumed to have been dropped, and then incremented by one for the packet that was retransmitted. Thus, decrementing the `pipe` by two packets when the first partial ACK is received is in some sense “cheating”, as that partial ACK only represents one packet having left the pipe. However, for any succeeding partial ACKs, `pipe` was incremented when the retransmitted packet entered the pipe, but was never decremented for the packet assumed to have been dropped. Thus, when the succeeding partial ACK arrives, it does in fact represent two packets that have left the pipe: the original packet (assumed to have been dropped), and the retransmitted packet. Because the sender decrements `pipe` by two packets rather than one for partial ACKs, the SACK sender never recovers more slowly than a Slow-Start.

The `maxburst` parameter, which limits the number of packets that can be sent in response to a single incoming ACK packet, is experimental, and is not necessarily recommended for SACK implementations.[3]

There are a number of other proposals for TCP congestion control algorithms using selective acknowledgments [Kes94, MM96]. The SACK implementation in our simulator is designed to be the most conservative extension of the Reno congestion control algorithms, in that it makes the minimum changes to Reno's existing congestion control algorithms.

6 Simulations

This section describes simulations from four scenarios, with from one to four packets dropped from a window of data. Each set of scenarios is run for Tahoe, Reno, New-Reno, and SACK TCP. Following this section, Section 7 shows a trace of Reno TCP traffic taken from Internet traffic measurements, illustrating the performance problems of Reno TCP without SACK, and Section 8 discusses future directions of TCP with SACK.

For all of the TCP implementations in all of the scenarios, the first dropped packet is detected by the Fast Retransmit procedure, after the source receives three dup ACKs.

The results of the Tahoe simulations are similar in all four scenarios. The Tahoe sender recovers with a

[2] Our simulator simply works in units of packets, not in units of bytes or segments, and all data packets for a particular TCP connection are constrained to be the same size. Also note that a more aggressive implementation might decrement the variable `pipe` by more than one packet when an ACK packet with a SACK option is received reporting that the receiver has received more than one new out-of-order packet.

[3] For those reading the SACK code in the simulator, the boolean `overhead` parameter significantly complicates the code, but is only of concern in the simulator. The `overhead` parameter indicates whether some randomization should be added to the timing of the TCP connection. For all of the simulations in this paper, the `overhead` parameter is set to zero, implying no randomization is added.

Fast Retransmit followed by Slow-Start regardless of the number of packets dropped from the window of data. For connections with a larger congestion window, Tahoe's delay in slow-starting back up to half the previous congestion window can have a significant impact on overall performance.

The Reno implementation without SACK gives optimal performance when a single packet is dropped from a window of data. For the scenario in Figure 3 with two dropped packets, the sender goes through Fast Retransmit and Fast Recovery twice in succession, unnecessarily reducing the congestion window twice. For the scenarios with three or four packet drops, the Reno sender has to wait for a retransmit timer to recover.

As expected, the New-Reno and SACK TCPs each recover from all four scenarios without having to wait for a retransmit timeout. The New-Reno and SACK TCPs simulations look quite similar. However, the New-Reno sender is able to retransmit at most one dropped packet each round-trip time. The limitations of New-Reno, relative to SACK TCP, are more pronounced in scenarios with larger congestion windows and a larger number of dropped packets from a window of data. In this case the constraint of retransmitting at most one dropped packet each round-trip time results in substantial delay in retransmitting the later dropped packets in the window. In addition, if the sender is limited by the receiver's advertised window during this recovery period, then the sender can be unable to effectively use the available bandwidth.[4].

For each of the four scenarios, the SACK sender recovers with good performance in both per-packet end-to-end delay and overall throughput.

6.1 The simulation scenario

The rest of this section consists of a detailed description of the simulations in Figures 2 through 5. All of these simulations can be run on our simulator `ns` with the command `test-sack`. For those readers who are interested, the text gives a packet-by-packet description of the behavior of TCP in each simulation.

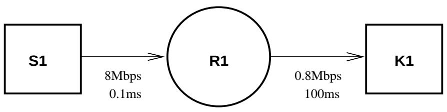

Figure 1: Simulation Topology

Figure 1 shows the network used for the simulations in this paper. The circle indicates a finite-buffer drop-tail gateway, and the squares indicate sending or receiving hosts. The links are labeled with their bandwidth capacity and delay. Each simulation has three TCP connections from S1 to K1. Only the first connection is shown in the figures. The second and third connections have limited data to send, and are included to achieve the desired pattern of packet drops for the first connection. The pattern of packet drops is changed simply by changing the number of packets sent by the second and third connections. Readers interested in the exact details of the simulation set-up are referred to the files `test-sack` and `sack.tcl` in our simulator `ns` [MF95]. The granularity of the TCP clock is set to 100 msec, giving round-trip time measurements accurate to only the nearest 100 msec.

These simulations use drop-tail gateways with small buffers. These are not intended to be realistic scenarios, or realistic values for the buffer size. They are intended as a simple scenario for illustrating TCP's congestion control algorithms. Simulations with RED (Random Early Detection) gateways [FJ93] would in general avoid the bursts of packet drops characteristic of drop-tail gateways.

`Ns` [MF95] is based on LBNL's previous simulator `tcpsim`, which was in turn based on the REAL simulator [Kes88]. The simulator does not use production TCP code, and does not pretend to reproduce the exact behavior of specific implementations of TCP [Flo95]. Instead, the simulator is intended to support exploration of underlying TCP congestion and error control algorithms, including Slow-Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery. The simulation results contained in this report can be recreated with the `test-sack` script supplied with `ns`.

For simplicity, most of the simulations shown in this paper use a data receiver that sends an ACK for every data packet received. The simulations in this paper also consist of one-way traffic. As a result, ACKs are never "compressed" or discarded on the path from the receiver back to the sender. The simulation set run by the `test-sack` script includes simulations with multiple connections, two-way traffic, and data receivers that send an ACK for every two data packets received.

The graphs from the simulations were generated by tracing packets entering and departing from $R1$. For each graph, the x-axis shows the packet arrival or departure time in seconds. The y-axis shows the packet number mod 100. Packets are numbered starting with packet 0. Each packet arrival and departure is marked by a square on the graph. For example, a single packet passing through $R1$ experiencing no appreciable queueing delay would generate two marks so close together on the graph as to appear as a single mark. Packets delayed at $R1$ but not dropped will generate two colinear marks for a constant packet number, spaced by the queueing

[4]This is shown in the LBNL simulator `ns` in the test `many-drops`, run with the command `test-sack`

delay. Packets dropped due to buffer overflow are indicated by an "×" on the graph for each packet dropped. Returning ACK packets received at $R1$ are marked by a smaller dot.

6.2 One Packet Loss

Figure 2 shows Tahoe, Reno, New-Reno, and SACK TCP with one dropped packet. Figure 2 shows that Tahoe requires a Slow-Start to recover from the packet drop, while Reno, New-Reno, and SACK TCP are all able to recover smoothly using Fast Recovery. The rest of this section describes the simulations in Figure 2 in more detail.

In Figure 2 with **Tahoe TCP**, packets 0–13 are sent without error as the sending TCP's congestion window increases exponentially from 1 to 15 according to the Slow-Start algorithm. The figure contains a square for each packet as it arrives and leaves the congested gateway. For a packet like the first one that experiences no queueing delay, the two squares appear as a single mark. As the queueing delay at the congested gateway increases, due in part to competing traffic not shown in this figure, the two marks for the arrival and departure diverge, and the distance between the arrival and departure marks corresponds to the queueing delay experienced by that packet.

By the end of the fourth non-overlapping window of data, the router's queue is full, causing packet 14 to be dropped. Because the first seven packets of the fourth window were successfully delivered (and ACKs are never dropped in these simulations), as the seven ACKs arrive the sender increases its window from 8 to 15 and sends the next 14 packets, 15–28.

After receiving the first ACK for packet 13, the sender receives 14 additional ACKs for packet 13 corresponding to the receiver's successful receipt of packets 15–28. The third duplicate ACK of the sequence (the fourth ACK for packet 13) meets the duplicate ACK threshold of three, and Fast Retransmission and Slow-Start are invoked. In addition, the Slow-Start threshold *ssthresh*[5] is reduced to seven ($\lfloor \frac{8+7}{2} \rfloor$). The sending TCP resets its congestion window to one and retransmits packet 14.

The receiver has already cached packets 15–28, and upon receiving the retransmitted packet 14 acknowledges packet 28. The ACK for packet 28 causes the sender to increase its congestion window by one and continue its transmissions from packet 29. While transmitting the window beginning with packet 35, the sender reaches the Slow-Start threshold and enters Congestion Avoidance. During subsequent transmissions, the sender's window is increased by roughly one packet per round-trip time as expected.

For figure 2 with **Reno TCP**, Reno's Fast Recovery algorithm gives optimal performance in this scenario. The sender's congestion window is reduced by half, incoming dup acks are used to clock outgoing packets, and Slow-Start is avoided.

Reno's operation in Figure 2 is identical to Tahoe until the fourth ACK for packet 13 is received at the sender. The ACKs corresponding to packets 15–28 comprise 14 dup ACKs for packet 13. The third dup ACK triggers a retransmission of packet 14, puts the sender into Fast Recovery, and reduces its congestion window and Slow-Start threshold to seven. During Fast Recovery, receipt of the fourth dup ACK brings the usable window to 11, and by the 14th dup ACK the usable window reaches 21. The "inflated" window from the last six dup acks allows the sender to send packets 29–34. Upon receiving the ACK for packet 28, the sender exits Fast Recovery and continues in Congestion Avoidance with a congestion window of seven.

The **New-Reno** and **SACK** simulations in Figure 2 show no differences from the Reno simulation under one packet drop.

[5] The Slow-Start threshold *ssthresh* is a dynamically-set value indicating an upper bound on the congestion window above which a TCP sender switches from Slow-Start to the Congestion Avoidance algorithm.

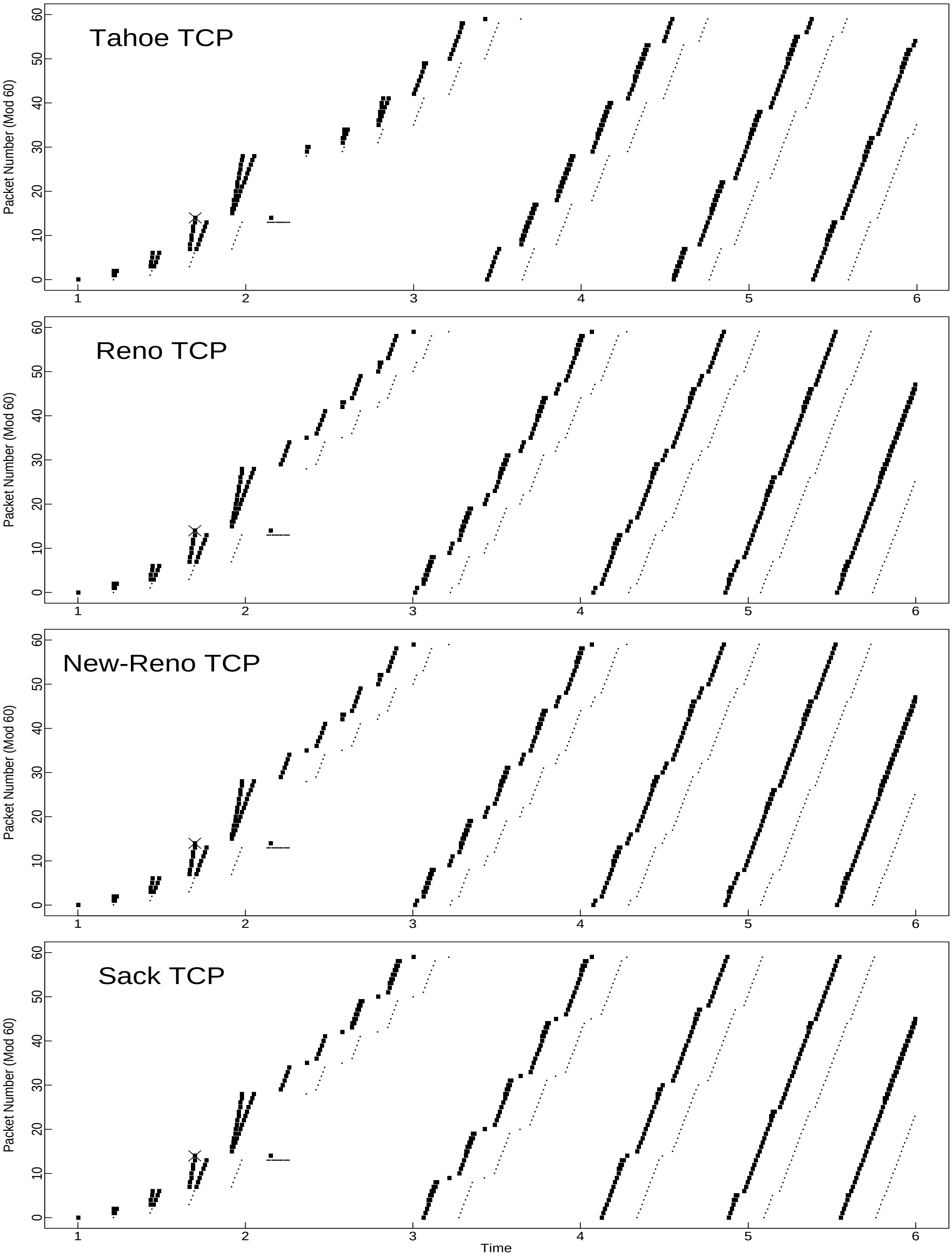

Figure 2: Simulations with one dropped packet.

6.3 Two Packet Losses

Figure 3 shows Tahoe, Reno, New-Reno, and SACK TCP with two dropped packets. As in the previous simulation, Tahoe recovers from the packet drops with a Slow-Start. Reno TCP recovers with some difficulties, while both New-Reno and SACK TCP recover smoothly and quickly. The rest of this section describes the simulations in Figure 3 in more detail.

The top figure in Figure 3 shows **Tahoe TCP** with two dropped packets. The response to loss on packet 14 is as described for Tahoe in the single loss case. In Tahoe, even though packets 15–28 were sent, this fact is forgotten by the sender when retransmitting packet 14.

After retransmitting packet 14 and receiving 13 dup ACKs, the sender receives an ACK for packet 27. The sender is in Slow-Start, opens its window to 2, and sends packets 28 and 29. The sender switches from Slow-Start to Congestion Avoidance when sending packet 40.

The Reno sender is often forced to wait for a retransmit timeout to recover from two packets dropped from a window of data.[6] In Figure 3 with **Reno TCP**'s Fast Retransmit, the Reno sender does not have to wait for a retransmit timeout, but instead recovers by doing a Fast Retransmit and Fast Recovery two times in succession, in the process cutting the congestion window in half twice, in two successive round-trip times. This slows down the TCP connection considerably.

The two packet drops occur at packets 14 and 28. Operation is similar to the one-drop case, except the loss of packet 28 implies 13 dup ACKs are generated for packet 13 rather than 14. The 13 dup ACKs allow the sender to send packets 29–33 with a usable window of 20 after the last dup ACK is received.

The loss of packet 28 causes a number of dup ACKs for packet 27 to be received at the sender. The first ACK for packet 27 is triggered by the receiver receiving the retransmitted packet 14. This ACK allows the sender to send packet 34. The next five dup ACKs are triggered by packets 29–33, and the final dup ACK is triggered by packet 34.

At the time the first ACK for packet 27 is received, the sender exits Fast Recovery with a congestion window of seven, having been reduced from 15 after the first loss. Upon receipt of the third dup ACK for packet 27, the sender begins a second Fast Retransmit. The sender retransmits packet 28 and reduces its congestion window to three, but is unable to send any additional data because of its usable window of six. The usable window grows from eight to nine upon receipt of the fifth and sixth dup ACKs, allowing the sender to send packets 35 and 36.

The sender receives an ACK for packet 34 as a result of the receiver receiving retransmitted packet 28. This ACK brings the sender out of Fast Recovery with a congestion window and *ssthresh* of three. The ACKs for packets 34 and 35 allow the sender to send 37 and 38, and the ACK for packet 36 allows packet 39 to be sent. The pattern repeats for many round-trip times, alternating between a single ACK advancing the sender's window followed by a series of ACKs which both advance and expand the sender's window according to Congestion Avoidance.

In figure 3 with **New-Reno TCP**, New-Reno's behavior is similar to Reno until the sender receives the first ACK for packet 27. This ACK is a partial ACK, and causes New-Reno to retransmit packet 28 immediately and not exit Fast Recovery. The dup ACK counter is reset to zero and later increased by the number of dup ACKs matching the partial ACK. The congestion window is not affected.

With the arrival of five dup ACKs for packet 27, the sender sends packets 35–39. The ACK for packet 33 causes the sender to exit Fast Recovery with a congestion window of seven and continue in Congestion Avoidance.

In figure 3 with **SACK TCP**, SACK TCP's behavior is similar to Reno until the sender receives the third ACK for packet 13. At this point, the protocol initializes the `pipe` as follows:

$$pipe = cwnd - ndup = 15 - 3 = 12.$$

It then subtracts one for each of the subsequent 10 dup ACKs and adds one for each of the five transmitted packets 29–33. At the point the first ACK for packet 27 arrives, `pipe` has value $12 - 10 + 5 = 7$.

The first ACK for packet 27 is a partial ACK, causing `pipe` to be decremented by two. With the sender's congestion window at seven, packets 34 and 35 are now sent. The five additional dup ACKs for packet 27 minus one for the retransmission of packet 28 allow the sender to send packets 36–39. The sender next receives two dup ACKs for packet 27 corresponding to the receipt of packets 34 and 35, allowing the sender to send packets 40 and 41. The next ACK received at the sender is for packet 35 and corresponds to the receiver receiving the retransmitted packet 28. It brings the sender out of Fast Recovery with a congestion window of seven, thereby allowing packet 42 to be sent. The next four ACKs for packets 36–39 allow the sender to send packets 43–46 and continue under Congestion Avoidance.

[6] More precisely, when two packets are dropped from a window of data, the Reno sender is forced to wait for a retransmit timeout whenever the congestion window is less than 10 packets when Fast Recovery is initiated, and whenever the congestion window is within two packets of the receiver's advertised window when Fast Recovery is initiated.

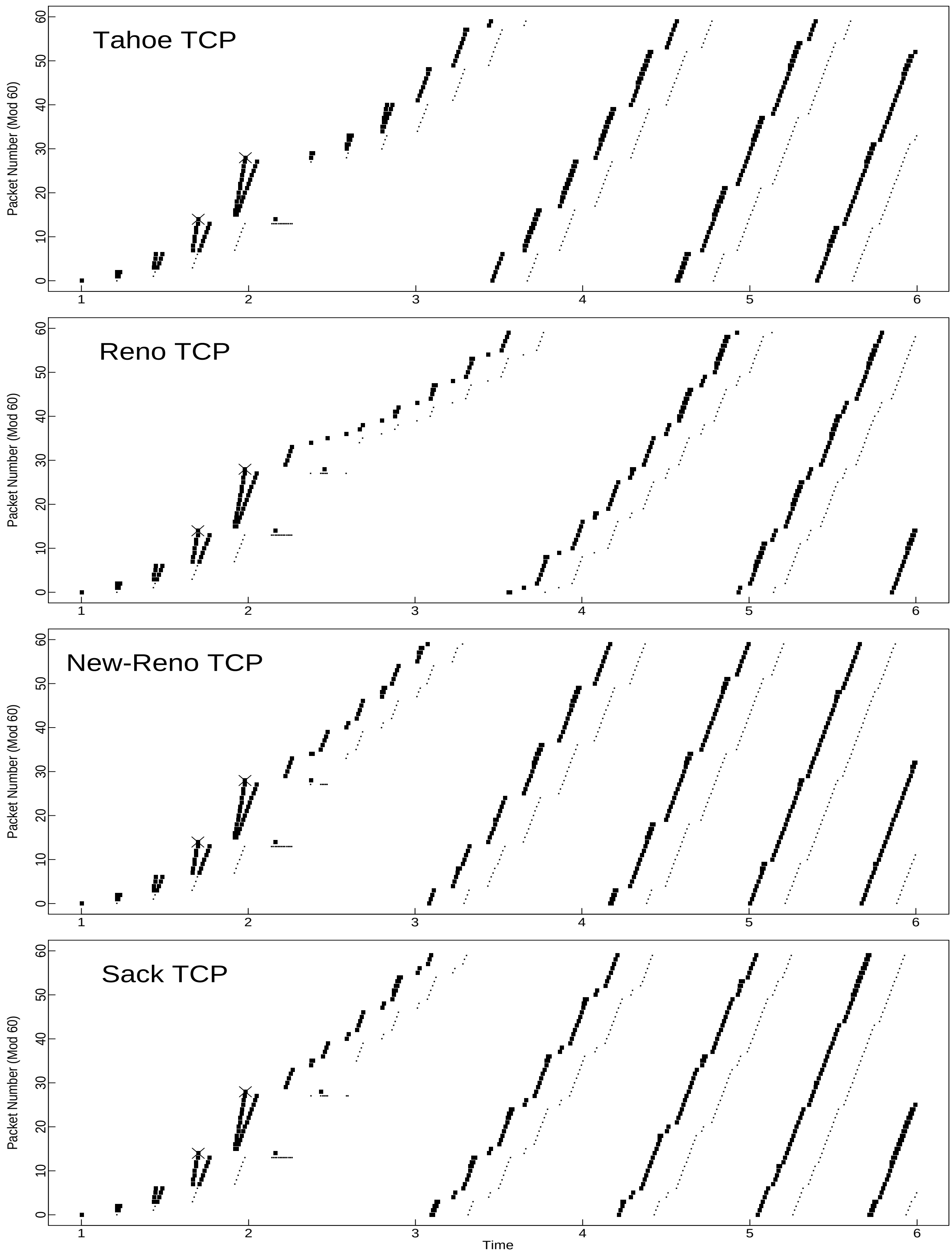

Figure 3: Simulations with two dropped packets.

6.4 Three Packet Losses

Figure 4 shows Tahoe, Reno, New-Reno, and SACK TCP with three dropped packets. As in the previous simulations, Tahoe recovers from the packet drops with a Slow-Start. Reno TCP, on the other hand, experiences severe performance problems, and has to wait for a retransmit timer to recover from the dropped packets. Both New-Reno and SACK TCP recover fairly smoothly. The rest of this section describes the simulations in Figure 4 in more detail.

The top figure in Figure 4 shows **Tahoe TCP** with three dropped packets. The response to loss on packet 14 is as described for Tahoe in the single loss case. As in the two packet loss case, even though packets 15–28 were sent, this is not taken into account by the sender.

After retransmitting packet 14 and receiving 12 dup ACKs, the sender receives an ACK for packet 25. The sender is in Slow-Start, opens its window to 2, and sends packets 26 and 27. Note that packets 26 and 27 are sent a second time, even though 27 has already been successfully received. The sender next receives two ACKs for packet 27, corresponding to the receipt of the resent packets 26 and 27. One of these ACKs is for new data, which increases the congestion window to three. The sender continues in Slow-Start until packet 37, where it switches to Congestion Avoidance.

Figure 4 shows **Reno TCP** with three dropped packets. When three packets are dropped from a window of data, the Reno sender is almost always forced to wait for a retransmit timeout.[7]

Reno's operation in Figure 4 is generally similar to Reno with two drops, except the additional packet drop causes only 12 dup ACKs for packet 13 rather than thirteen. The 12 dup ACKs allow the sender to send packet 29–32 with a usable window of 19 after retransmitting packet 14.

With the arrival of the first ACK for packet 25, Reno exits Fast Recovery, but after receiving three additional ACKs re-enters Fast Recovery with a congestion window of three and usable window of six. With the arrival of the fifth ACK for packet 25, the usable window grows to seven, but the sender is still unable to send data because seven packets (26–32) are still unacknowledged. The ACK for packet 27 brings the sender out of Fast Recovery once again with a congestion window of three. At the point the ACK for packet 27 arrives, the sender is stalled. Although packets 28–32 have not yet been acknowledged and 28 requires retransmission, the "ACK clock" is lost, implying Reno is unable to employ Fast Retransmit and must instead await a retransmission timeout.

The timeout for packet 28 expires, causing a retransmission and putting the sender into Slow-Start. The ACK for packet 32 corresponds to the arrival of packet 28 at the receiver, and the sender continues in Congestion Avoidance as expected.

Figure 4 shows **New-Reno TCP** with three dropped packets. New-Reno's operation is similar to Reno with three drops until the receipt of the first ACK for packet 25. After receiving this ACK, the New-Reno sender immediately retransmits packet 26 and sets its usable window to a congestion window of seven. The four subsequent dup ACKs for packet 25 inflate the usable window to eleven, allowing the sender to send packets 33–36. The next partial ACK acknowledges packet 27 and causes the sender to retransmit packet 28 and reduce its usable window to seven. The sender is unable to send additional data until the receipt of the third and fourth dup ACKs for packet 27, which allow the sender to send packets 37 and 38 with a usable window of eleven.

The ACK for packet 36 brings the sender out of Fast Recovery and returns its congestion window to seven. Only packets 37 and 38 are unacknowledged at this point, so the sender should be able to send five additional packets but is instead limited to sending only four packets by the `maxburst` parameter described above. The arrival of the ACKs for packets 37 and 38 allows the sender to send packets 43 and 44 followed by 45, respectively. The sender continues in Congestion Avoidance with a window of seven.

Figure 4 shows **SACK TCP** with three dropped packets. SACK TCP's packet sending pattern is similar to Reno with three packet drops, until the 12th dup ACK for packet 13 is received at the sender. This ACK contains SACK information indicating a "hole" at packet 26. Rather than sending packets 29–32 as in Reno, it instead sends 29–31 and retransmits 26.

The handling of `pipe` is similar to SACK TCP with two packet drops. When the third dup ACK for packet 13 arrives at the sender, `pipe` is initialized to 12. The retransmission of packet 26 is accounted for, causing the value of `pipe` to become $12 - 9 + 1 + 3 = 7$ when the first ACK for packet 25 arrives. This ACK corresponds to the receiver receiving the retransmitted packet 14, and causes the sender to reduce `pipe` by two and send packets 32 and 33.

[7] When three packets are dropped from a window of data, the Reno sender is forced to wait for a retransmit timeout whenever the number of packets between the first and the second dropped packets is less than $2 + 3W/4$, for W the congestion window just before the Fast Retransmit.

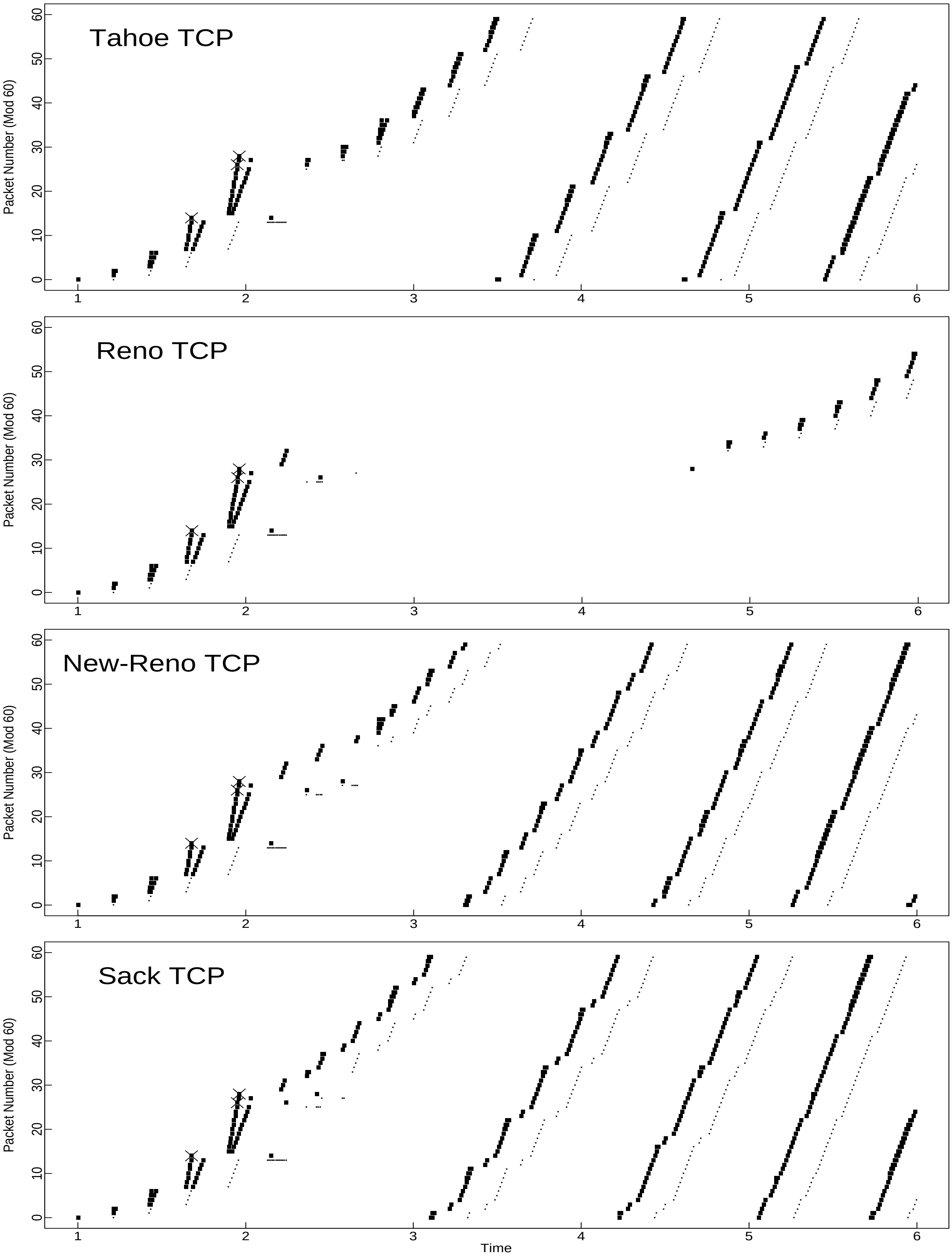

Figure 4: Simulations with three dropped packets.

The next three ACKs acknowledge packet 25 and contain SACK information indicating a hole at packets 26 and 28. The three ACKs cause the sender to reduce `pipe` by three and retransmit packet 28. At that point no holes remain to be filled and the sender may send packets 34 and 35. The next ACK arrives shortly thereafter, acknowledges packet 27 and indicates the hole at packet 28. It is also a partial ACK, causing `pipe` to be decremented by two and allowing the sender to send packets 36 and 37.

The next two ACKs for packet 27 arrive nearly together and correspond to the receiver receiving packets 32 and 33. These ACKs contain SACK information indicating the hole at packet 28 remains to be filled. As the sender has already retransmitted 28 and no other holes are indicated in the SACK information, the sender continues by sending packets 38 and 39. The next ACK received at the sender corresponds to the receiver's receipt of the retransmission of packet 28. It acknowledges packet 33 and brings the sender out of Fast Recovery with a congestion window of 7. The sender continues in Congestion Avoidance.

6.5 Four Packet Losses

Figure 5 shows Tahoe, Reno, New-Reno, and SACK TCP with four dropped packets. As in the previous simulations, Tahoe recovers from the packet drops with a Slow-Start. Also as in the previous simulation, Reno TCP experiences severe performance problems, and has to wait for a retransmit timer to recover from the dropped packets. New-Reno requires four round-trip times to recover and to retransmit the four dropped packets, while the SACK TCP sender recovers quickly and smoothly. The differences between New-Reno and SACK TCP become more pronounced if even more packets are dropped from the window of data. The rest of this section describes the simulations in Figure 5 in more detail.

The top figure in Figure 5 shows **Tahoe TCP** with four dropped packets. The response to loss on packet 14 is as described for Tahoe in the single loss case. Once again, the transmission of packets 15–28 is forgotten by the sender when retransmitting packet 14.

After retransmitting packet 14 and receiving 11 dup ACKs, the sender receives an ACK for packet 23. The sender is in Slow-Start, opens its window to 2, and sends packets 24 and 25. Once again, Tahoe duplicates effort on packet 25.

The sender next receives two ACKs for packet 25, corresponding to receipt of the resent packets 24 and 25. One of these ACKs is for new data, which increases the congestion window to three. The sender then sends packets 26–28, again duplicating effort on packet 27. The next pair of ACKs, one for new data and one duplicate, correspond to the receiver's receipt of packets 26 and 27 and increase the sender's congestion window to four. The ACK for packet 28 arrives next, increases the congestion widow to five, and continues in Slow-Start. The sender switches to Congestion Avoidance as it sends packet 35 and continues in Congestion Avoidance as expected.

For Figure 5 with **Reno TCP**, the sender is always forced to wait for a retransmit timeout when four packets are dropped from a single window of data.

The sender receives eleven dup ACKs for packet 14, retransmits packet 14 on the third and is able to send packets 29–31 as a result of receiving the ninth through eleventh dup ACKs. The ACK for packet 23 brings the sender out of Fast Recovery with a usable window set to the congestion window of seven. The third dup ACK, corresponding to the receiver's receipt of packets 29–31, initiates a second Fast Retransmit and Fast Recovery, triggering a retransmission of packet 24, reducing the congestion window to three, and setting the usable window to six. As packets 24–31 are unacknowledged, the sender cannot proceed until it receives another ACK.

The next ACK for packet 25 brings the sender out of Fast Recovery again, bringing the congestion window and usable window to three. As in the case of three drops, the sender is frozen because the six unacknowledged packets exceeds the congestion window and the ACK clock is lost. The sender must await a retransmission timer expiration to proceed.

Once the timer expires, the sender retransmits packet 26, receives an ACK for packet 27, and transmits 28 and 29. After a timer expiration, Reno behaves similarly to Tahoe, in that it sometimes retransmits packets (in this case, packet 29) that it has already transmitted and that have already been cached at the receiver. After receiving two ACKs for packet 31 it continues in Congestion Avoidance.

In Figure 5 with **New-Reno TCP**, New-Reno's operation is similar to Reno with three drops until the receipt of the first ACK for packet 23. Upon receiving this ACK, the sender immediately retransmits packet 24 and sets its usable window to the congestion window of seven. The three subsequent dup ACKs for packet 23 inflate the usable window to ten, allowing the sender to send packets 32 and 33. The next partial ACK acknowledges packet 25 and causes the sender to retransmit packet 26 and reduce its usable window to seven.

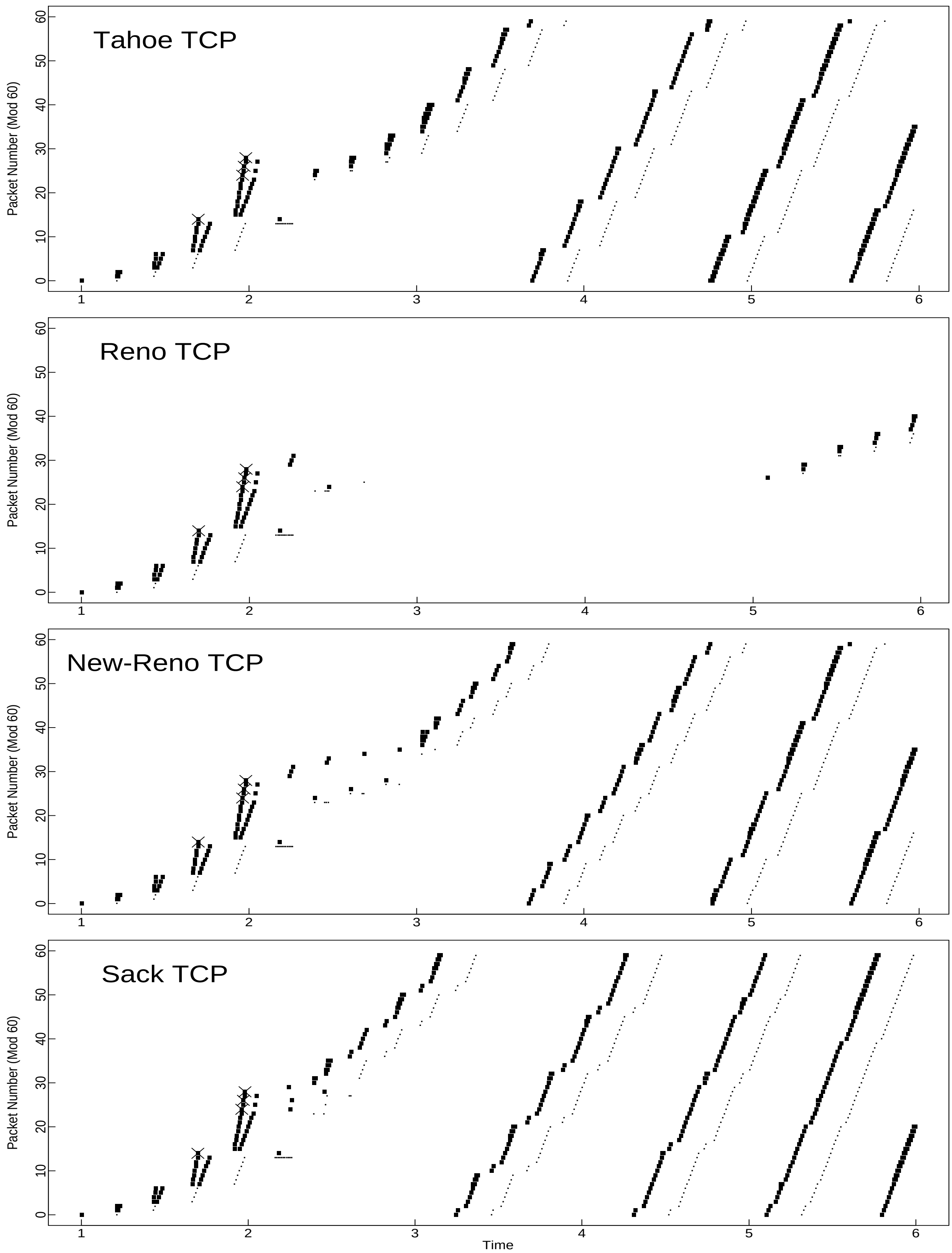

Figure 5: Simulations with four dropped packets.

The sender is unable to send additional data until the receipt of the second dup ACKs for packet 25, which allows the sender to send packet 34 with a usable window of nine. The last partial ACK acknowledges packet 27 and causes the sender to retransmit packet 28 and reduce its usable window to seven. The sender is again unable to send additional data until the receipt of the dup ACK for packet 27, which allows the sender to send packet 35 with a usable window of eight.

The ACK for packet 34 brings the sender out of Fast Recovery and returns its congestion window to seven. Only packet 35 is unacknowledged at this point, so the sender should be able to send six additional packets but is instead limited to sending only four by the "maxburst" parameter described above. The arrival of the ACK for packet 35 allows the sender to send packets 40–42. The sender continues in Congestion Avoidance with a window of seven.

In Figure 5 with **SACK TCP**, SACK TCP's packet sending pattern is similar to Reno with four packet drops, until the 10th dup ACK for packet 13 is received at the sender indicating a hole at packet 24. The 11th dup ACK for packet 13 indicates holes at packets 24 and 26. The sender retransmits packets 24 and 26 as a result of these ACKs.

The handling of `pipe` is similar to SACK TCP with three packet drops. When the third dup ACK for packet 13 arrives at the sender, `pipe` is initialized to 12. The retransmission of packets 24 and 26 are accounted for, causing the value of `pipe` to be $12-8+2+1=7$ when the first ACK for packet 23 arrives. This partial ACK, corresponding to the receiver receiving the retransmitted packet 14, causes the sender to reduce `pipe` by two, and also contains SACK information indicating holes at packets 24 and 26. The sender proceeds by sending packets 30 and 31 because 24 and 26 have already been retransmitted.

The dup ACK for packet 23 corresponds to the receiver receiving packet 29 and contains SACK information indicating holes at packets 24, 26 and 28. Again the sender notices it has already retransmitted 24 and 26, and thus proceeds by retransmitting 28. A short time later an ACK for packet 25 arrives, indicating the holes at packets 26 and 28. The ACK for packet 27 arrives next, indicating the hole at packet 28. Each of these ACKs reduces pipe by two, allowing the sender to send packets 32–35 because it has already retransmitted 28.

The next two ACKs for packet 27 arrive nearly together and correspond to the receiver receiving packets 30 and 31. These ACKs contain SACK information indicating the hole at packet 28 remains to be filled. Once again, the sender avoids retransmitting packet 28 and continues by sending packets 36 and 37. The next ACK received at the sender corresponds to the receiver's receipt of the retransmission of packet 28. It acknowledges packet 31 and brings the sender out of Fast Recovery with a congestion window of 7. The sender continues in Congestion Avoidance.

7 A trace of Reno TCP

The TCP trace in this section is taken from actual Internet traffic measurements, but exhibits behavior similar to that in our simulator. It shows the poor performance of Reno without SACK when multiple packets are dropped from one window of data. The TCP connection in this trace repeated has two packets dropped from a window of data, and each time is forced to wait for a retransmit timeout to recover.

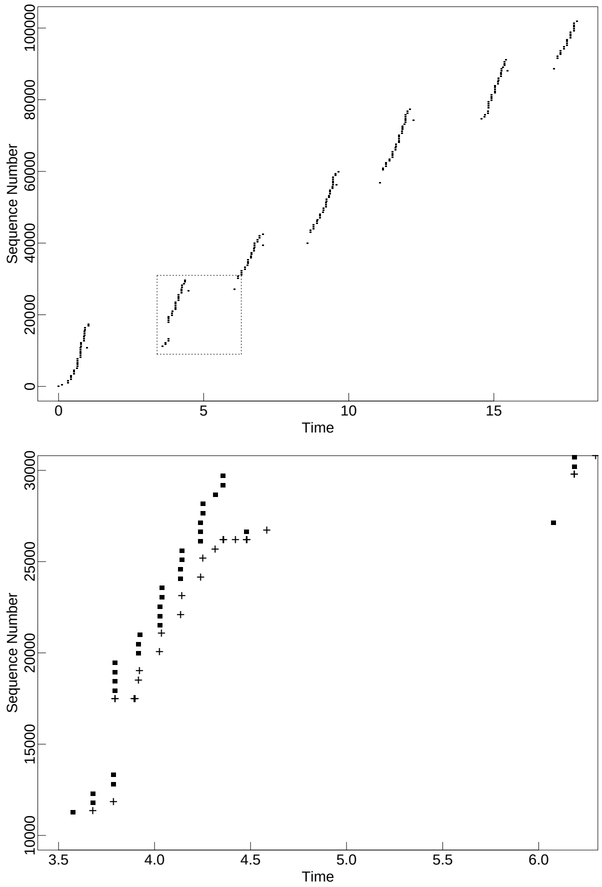

Figure 6: A trace of Reno TCP.

The trace in Figure 6 shows a TCP connection from the San Diego Supercomputer Center (SDSC) in San Diego, using IRIX-5.2, to Brookhaven National Laboratory on Long Island, using IRIX-5.1.1. The TCP connection receives poor throughput because of repeated waits for a retransmit timeout. The graph on the right gives a enlargement of a section from the graph on the left. The blowup shows a mark for every packet transmitted, and a "+" for every ACK received.

The enlargement shows that the data receiver uses a delayed-ACK algorithm, usually sending a single ACK for every two data packets. As a result, in the Congestion Avoidance phase the data sender normally sends two data packets for every ACK packet received. When an ACK packet is received that causes the sender to increase its congestion window by one packet, then the data sender sends three data packets after receiving a single ACK packet. As an example, at time 4.24 the data sender receives an ACK acknowledging sequence number 24065, and the data sender sends three packets, for sequence numbers 26113-27648. The last two of the three packets are dropped.

At time 4.48 the data sender receives a third dup ACK (in the figure this is printed on top of the second dup ACK), executes Fast Retransmit, retransmits one packet, and later receives an ACK for that packet. However, at this point the sender's congestion window is half of its old value, and this is not large enough to permit the sender to send the next highest packet. The sender waits for a retransmit timer to expire before retransmitting the second packet that was dropped from the original window of data. This is similar to the Reno behavior illustrated in the simulator. This is an example of a scenario where Tahoe might give better performance that Reno.

The trace was supplied by Vern Paxson, as part of work on his Ph.D. thesis. Vern reports that 13% of his 2299 collected TCP traces show this behavior. That is, 13% of his TCP traces contain a Fast Retransmit followed by a retransmit timeout, where the packet retransmitted after the retransmit timeout had not been previously retransmitted by the TCP sender. This additional condition eliminates incidents from Tahoe or Reno traces where the retransmit timeout is required simply because a retransmitted packet is itself dropped. Thus, 13% of Vern's TCP traces are likely to include Reno TCP with multiple packet drops and an unnecessary retransmit timeout.

8 Future directions for selective acknowledgments

The addition of selective acknowledgments allows additional improvements to TCP, in addition to improving the congestion control behavior when multiple packets are dropped in one window of data. [MM96] explores TCP congestion control algorithms for TCP with SACK. [BPSK96] shows that SACK and explicit wireless loss notification both result in substantial performance improvements for TCP over lossy links. Several researchers are exploring the use of SACK, coupled with the explicit notification of non-congestion-related losses, for lossy environments such as satellite links.

The SACK option will allow the TCP protocol to be more intelligent in other ways as well.[8] As one example, the use of selective acknowledgments will allow the sender to make a more intelligent response to the first or second dup ACKs. Most TCP implementations, including the ones shown in this paper, simply ignore the first or second dup ACKs. With SACK, the sender will know if a dup ACK indicates that another packet has in fact left the pipe, allowing the sender to send a new packet if the receiver's advertised window permits. Further, with SACK the sender will know *which* packet has left the network, allowing the sender to make an informed guess about whether this is likely to be the last dup ACK that it will receive.

As a second example, by giving precise information on the exact data received by the receiver, and the order in which that data was received, the use of SACK would allow the sender to infer when it has mistakenly assumed that a packet was dropped, and therefore to rescind its decision to reduce the congestion window.

As a third example, by effectively decoupling decisions of *when* to send a packet from decisions of *which* packet to send, SACK opens the way to further advances of TCP's congestion control algorithms.

The SACK implementation in our simulator could be improved in its robustness to reordered packets during Fast Recovery. If, during Fast Recovery, the sender receives a SACK packet with a SACK block for packet n, and a second SACK block repeating a report for packet $n-2$, the sender in our implementation might immediately retransmit packet $n-1$. Probably the sender should wait for a few more ACKs all indicating that packet $n-1$ is missing at the receiver, to give robustness against reordered packets.

The New-Reno and SACK implementations in our simulator use a "maxburst" parameter to limit the potential burstiness of the sender for the first window of packets sent after exiting from Fast Recovery. This is mainly an issue when the sender has been prevented from sending packets during Fast Recovery because of restrictions imposed by the receiver's advertised window. An improved SACK implementation would only use a "maxburst" parameter immediately after leaving Fast Recovery. A comparable mechanism to prevent bursts would be, upon exiting Fast Recovery, to set the congestion window to the number of packets known to be in the pipe, to set *ssthresh* to what would have been the congestion window, and to use Slow-Start to quickly

[8] These proposals are not necessarily original with us, but are from general discussions in the research community about the use of SACK. Unfortunately, we don't have a precise attribution for each proposal.

increase the congestion window back up to *ssthresh*.

9 Conclusions

In this paper we have explored the fundamental restrictions imposed by the lack of selective acknowledgments in TCP, and have examined a TCP implementation that incorporates selective acknowledgments into Reno TCP while making minimal changes to TCP's underlying congestion control algorithms. We assume that the addition of selective acknowledgments to TCP will open the way to further developments of the TCP protocol.

10 Acknowledgements

This document[9] was written in support of [MMFR96], the current proposal for adding a SACK option to TCP, and draws from discussions about SACK and TCP with a wide range of people. We would in particular like to thank Hari Balakrishnan, Bob Braden, Janey Hoe, Van Jacobson, Jamshid Mahdavi, Matt Mathis, Vern Paxson, Allyn Romanow, and Lixia Zhang. We thank Vern Paxson for the TCP traces. The implementation of SACK TCP in the simulator is in large part from Matt Mathis and Jamshid Mahdavi.

[9] The earlier versions of this note are available at URL ftp://ftp.ee.lbl.gov/papers/sacks_v0.ps.Z (December 1995) and URL ftp://ftp.ee.lbl.gov/papers/sacks_v1.ps.Z (March 1996). While the results are essentially unchanged, the earlier results used non-standard TCP implementations where the sender's maximum congestion window is assumed to be less than the receiver's advertised window.

References

[BBJ92] D. Borman, R. Braden, and V. Jacobson. "TCP Extensions for High Performance,". Request for Comments (Proposed Standard) RFC 1323, Internet Engineering Task Force, May 1992. (Obsoletes RFC1185).

[BJ88] R. Braden and V. Jacobson. "TCP extensions for long-delay paths,". Request for Comments (Experimental) RFC 1072, Internet Engineering Task Force, October 1988.

[BJZ90] R. Braden, V. Jacobson, and L. Zhang. "TCP Extension for High-Speed Paths,". Request for Comments (Experimental) RFC 1185, Internet Engineering Task Force, October 1990. (Obsoleted by RFC1323).

[BPSK96] H. Balakrishnan, V.N. Padmanabhan, S. Seshan, and R.H. Katz. "A Comparison of Mechanisms for Improving TCP Performance over Wireless Links,". *SIGCOMM Symposium on Communications Architectures and Protocols*, Aug. 1996. to appear.

[Bra94] R. Braden. "T/TCP – TCP Extensions for Transactions Functional Specification,". Request for Comments (Experimental) RFC 1644, Internet Engineering Task Force, July 1994.

[CH95] D.D. Clark and J. Hoe. "Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes,". Technical report, Jun. 1995. Presentation to the Internet End-to-End Research Group, cited for acknowledgement purposes only.

[Che88] D. Cheriton. "VMTP: Versatile Message Transaction Protocol: Protocol specification,". Request for Comments (Experimental) RFC 1045, Internet Engineering Task Force, February 1988.

[CLZ87] D. Clark, M. Lambert, and L. Zhang. "NETBLT: A bulk data transfer protocol,". Request for Comments (Experimental) RFC 998, Internet Engineering Task Force, March 1987. (Obsoletes RFC0969).

[FJ93] Sally Floyd and Van Jacobson. "Random Early Detection Gateways for Congestion Avoidance,". *IEEE/ACM Transactions on Networking*, 1(4):397–413, Aug. 1993. URL http://www-nrg.ee.lbl.gov/nrg-papers.html.

[Flo95] Sally Floyd. "Simulator Tests,". Technical report, Jul. 1995. URL http://www-nrg.ee.lbl.gov/nrg-papers.html.

[Flo96a] S. Floyd. "Issues of TCP with SACK,". Technical report, Mar. 1996. URL ftp://ftp.ee.lbl.gov/papers/issues_sa.ps.Z.

[Flo96b] S. Floyd. "SACK TCP: The sender's congestion control algorithms for the implementation "sack1" in LBNL's "ns" simulator (viewgraphs).,". Technical report, Mar. 1996. Presentation to the TCP Large Windows Working Group of the IETF, March 7, 1996. URL ftp://ftp.ee.lbl.gov/talks/sacks.ps.

[Hoe95] J. Hoe. “Start-up Dynamics of TCP's Congestion Control and Avoidance Schemes,”. Jun. 1995. Master's thesis, MIT.

[Hoe96] J. Hoe. “Improving the Start-up Behavior of a Congestion Control Scheme for TCP,”. *SIGCOMM Symposium on Communications Architectures and Protocols*, Aug. 1996. to appear.

[HSV84] R. Hinden, J. Sax, and D. Velten. “Reliable Data Protocol,”. Request for Comments (Experimental) RFC 908, Internet Engineering Task Force, July 1984. (Updated by RFC1151).

[Jac88] V. Jacobson. “Congestion Avoidance and Control,”. *SIGCOMM Symposium on Communications Architectures and Protocols*, pages 314–329, 1988. An updated version is available via ftp://ftp.ee.lbl.gov/papers/congavoid.ps.Z.

[Jac90] V. Jacobson. “Modified TCP Congestion Avoidance Algorithm,”. Technical report, 30 Apr. 1990. Email to the end2end-interest Mailing List, URL ftp://ftp.ee.lbl.gov/email/vanj.90apr30.txt.

[Kes88] S. Keshav. “REAL: a Network Simulator,”. Technical Report 88/472, University of California Berkeley, Berkeley, California, 1988.

[Kes94] S. Keshav. “Packet-Pair Flow Control,”. Technical report, Nov. 1994. Presentation to the Internet End-to-End Research Group, cited for acknowledgement purposes only.

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

[MM96] Matthew Mathis and Jamshid Mahdavi. “Forward Acknowledgement: Refining TCP Congestion Control,”. *SIGCOMM Symposium on Communications Architectures and Protocols*, Aug. 1996. to appear.

[MMFR96] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. “TCP Selective Acknowledgment Options,”. (Internet draft, work in progress), 1996.

[SDW92] W. T. Strayer, B. Dempsey, and A. Weaver. *XTP: The Xpress Transfer Protocol*. Addison Wesley, Reading, MA, 1992.

[Ste94] W. Richard Stevens. *TCP/IP Illustrated, Volume 1: The Protocols*. Addison Wesley, 1994.