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 DRAFT

March 9, 1996

1 Introduction

The capability to provide selective acknowledgement (SACK) of received data has been used in a number of transport protocols including NETBLT [CLZ87], XTP [SDW92], RDP [HSV84] and VMTP [Che88]. Although there have been proposals for adding SACK to TCP [BJ88, BJZ90] SACK was later removed from the TCP RFCs (Request For Comments) [BBJ92] pending further research. In this report, we illustrate some of the benefits of adding SACK to "Reno" implementations of TCP. We use simulations to show that a SACK option with at most three SACK blocks can be of substantial benefit, relative to TCP without SACK. We have been running simulations of Reno TCP with SACK in our simulator at LBL for over five years [Flo91].

2 TCP Background and Terminology

Modern TCP implementations contain a number of algorithms aimed at controlling network congestion while maintaining good user throughput. Early TCP implementations follow a go-back-n model using cumulative positive acknowledgement 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, including *slow start*, *congestion avoidance*, and *fast retransmit*. With fast retransmit, a number of acknowledgements for the same sequence number (dup acks) triggers a retransmission without awaiting a timer expiration. The slow start and congestion avoidance algorithms have been discussed elsewhere [Jac88a].

The "Reno" TCP implementation modified the TCP sender logic to include a scheme known as *fast recovery*. In "Tahoe", the receipt of a small number of dup acks triggered a slow start, thus emptying the pipe unnecessarily. With fast recovery, rather than performing a slow-start after the receipt of dup acks, the congestion window is effectively set to half its previous value [Jac88b, Ste94].

This paper shows several simulations of Reno TCP that have been designed to highlight Reno's weaknesses. The main weakness of Reno TCP without SACK is that when multiple packets are dropped from one window of data, the sender often has to wait for a retransmit timer before recovering. We have illustrated this weakness with simulations that include all of the following ingredients: (1) drop-tail gateways (i.e., gateways that don't monitor the average queue size); (2) large-window TCP; (3) an end-to-end connection with a high delay-bandwidth product in terms of the number of packets required to fill the pipe; and (4) TCP connections that perform an initial slow-start into an empty pipe. In this case, the TCP source continues to double its congestion window each roundtrip time until the congestion window is quite large, the drop-tail buffer overflows, and a large number of packets are dropped from one window of data.

In our simulations we use three additional forms of TCP: Reno-SACK TCP, Reno-SACK1 TCP and New-Reno TCP. Both Reno-SACK and Reno-SACK1 implement the Reno algorithms with selective acknowledgements and selective retransmission.

The total size of a TCP header is limited to 60 bytes by the data offset field. A minimal TCP header comprises 20 bytes, leaving 40 to hold any options. Options are required to be 4-byte aligned. Presently, the recommended layout for sending the

*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.

TCP SACK option [RFC1072-1185bis, appendixA] is to include it following 2 bytes of padding and 10 bytes for time stamp option, leaving a total of 28 bytes for the SACK option. For n SACK blocks, $4n + 2$ bytes of header space are required— 2 bytes to describe the option, 2 for the each block offset, and 2 for each block size. Thus, ignoring the issue of scaled windows or of T/TCP [Bra94], six SACK blocks may be supported with two trailing padding bytes.

When operating with scaled windows, SACK offsets and sizes could either be scaled by the window scale factor and assumed to be exact, requiring a TCP sender to align its segments to the nearest segment size representable given the scale factor, or the sender could keep track of which segments have been sent and interpolate SACKs to infer which blocks failed to be received. An alternative to sender alignment and interpolation is to expand the number of bits used to represent the block offsets and sizes. If SACK block offsets and sizes are allowed to occupy twice as much header room, $8n + 2$ bytes of header space are required, providing support for three SACK blocks. The simulations in this note assume that this format is used. If T/TCP is used, T/TCP uses 8 bytes of option space (including padding). If both timestamps and T/TCP are used, then the option space has room for two SACK blocks.

The key performance problem of “Reno” TCP implementations without SACK is that, when multiple packets are dropped from one window of data, the TCP source often has to wait for a retransmit timer. However, this forced wait for the retransmit timer to expire is not a necessary consequence of the absence of the SACK option.

The purpose of the “New-Reno” TCP is to clarity the fundamental limitations of the absence of SACK. In current “Reno” TCP implementations, when multiple packets are dropped from one window of data, the TCP source often has to wait for a retransmit timer. More precisely, this wait for a retransmit timer sometimes occurs when two or three packets are dropped from one window of data, and always occurs when more than three packets are dropped. However, this wait for a retransmit timer is not a fundamental consequence of the absence of SACK. It is a fundamental consequence of the absence of SACK that the sender has to choose between either (1) retransmitting at most one dropped packet per roundtrip time, or (2) retransmitting packets that might have already been successfully received at the receiver.

The “New-Reno” TCP in this note includes a small change to the Reno algorithm at the sender that eliminates the wait for a retransmit timer when multiple packets are dropped from one window, at the expense of retransmitting at most one dropped packet per roundtrip time. The change concerns the sender's behavior during fast recovery when an ACK is received which acknowledges some but not all of the packets that were outstanding at the start of that fast recovery period; we call this a “partial ACK”. In Reno, partial ACKs take the TCP out of fast recovery. In New-Reno, partial ACKs do not trigger a change in the congestion window or take TCP out of fast recovery. Instead, in New-Reno partial ACKs during fast recovery are treated as an indication that the packet immediately following the acked packet in the sequence space has been lost, and should be retransmitted. [This idea was proposed by Janey Hoe in her recent master's thesis.] This difference between Reno and New-Reno TCP is illustrated by the simulations in Figure 4.

This note shows results for two different implementations of the SACK option. For the Reno-SACK simulations, the SACK option is assumed to have room for three SACK blocks, and these SACK blocks report the first three blocks in the sequence space. For the Reno-SACK1 simulations, the SACK option is assumed to have room for three SACK blocks, but the SACK blocks reported follow the guidelines in [MMFR95]. That is, the first SACK block reports the block containing the most recently received segment, and the additional SACK blocks repeat the most recently reported SACK blocks. This implementation, which contains key contributions from Matt Mathis and Jamshid Mahdavi, will be described further in future papers.

3 Simulation results of aggregate throughput

This section describes results from simulations exploring the relative link utilization achieved in simulations with Tahoe, Reno, New-Reno, and Reno-SACK TCP. Each simulation runs for eight seconds, on a network with 100Mbps access links, a 45Mbps shared link, Drop-Tail gateways with 50-packet queues, and TCP connections with a range of start times, packet sizes, and maximum windows. Each simulation scenario was run four times, with Tahoe, Reno, New-Reno, and Reno-SACK TCP. (That is, in the first run all of the TCP connections were using Tahoe TCP, in the second run all of the TCP connections were using Reno TCP, etc.)

For a simulation with a single TCP connection, multiple packets are often dropped during the initial slow-start, but after that, in the absence of delayed-ack receivers, one packet will be dropped during each congestion epoch. Thus, Tahoe and Reno-SACK generally perform better than Reno during the initial slow-start, but after that generally Reno and Reno-SACK will perform the same, and will both perform better than Tahoe. As an example of the range of results possible, in one eight-second scenario with a single Tahoe connection the link utilization is 87%, while with a Reno connection the link utilization is 96%. Changing the packet and window sizes can change the results so that the Tahoe connection gives a link utilization of 94% while a Reno connection gives a link utilization of only 84%. Both of these scenarios have high link utilization for a Reno-SACK connection with three SACK blocks.

For a simulation with a single TCP connection with delayed ack sinks, as illustrated in Figure 6, it is possible for the sender to send three packets back-to-back when the congestion window increases, and for two packets instead of one to be dropped in repeating congestion epochs. This is illustrated in an actual trace in Figure 6. In this case Tahoe can perform better than Reno even after the initial slow-start.

For this section, we have run a number of simulations of a scenario with several simultaneous TCP connections in each direction. We have created scenarios where Reno out-performs Tahoe, and scenarios where Tahoe significantly out-performs Reno. We have not been able to create a scenario where the link utilization is significantly different for Reno-SACK with six SACK blocks that it is for Reno-SACK with three SACK blocks.

4 Notes on the simulator

The simulator used for our tests is `ns` [MF95], based on LBL's previous simulator `tcpsim`, which was, in turn, based on the REAL simulator [Kes88]. The simulator does not use 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` command in ns.

For simplicity, most of the simulations in Section 6 don't use a delayed ACK receiver. Instead, the receiver sends an ACK for every packet. The simulations in Section 6 also consist of one-way traffic. As a result, acks are never discarded on the path from the receiver back to the sender. The simulations discussed in Section 3 include both two-way traffic and delayed-ACK receivers.

5 Simulation Setup

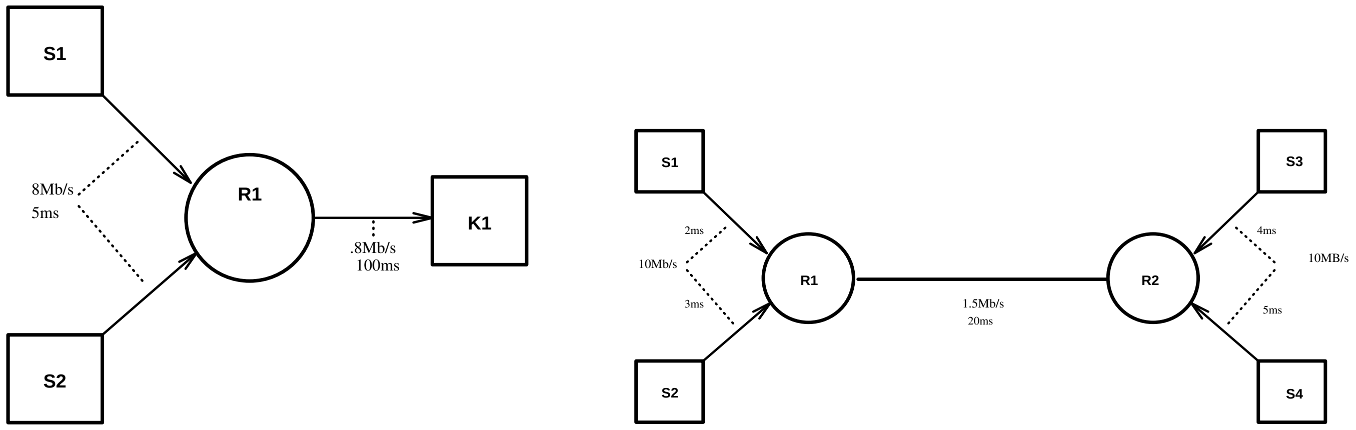

Figure 1: Simulated Topologies

Figure 1 illustrates the networks used for the simulations in Section 6. Circles indicate finite-buffer drop-tail gateways, and squares indicate sending or receiving hosts. The simple network on the left can be used to induce congestive loss by adjusting the maximum window size (e.g., the receiver's advertised window) for the TCP source at S1 (or S2). For all simulations discussed here, the granularity of the TCP clock is set to 100 msec, implying round-trip time measurements are only accurate to the nearest 100 msec.

These simulations use drop-tail gateways with 25-packet buffers. Simulations with RED (Random Early Detection) gateways [FJ93] would in general avoid the bursts of packet drops characteristic of drop-tail gateways.

6 Detailed simulation results

The simulations in this section are designed to highlight some of the differences between Tahoe, Reno, and Reno-SACK TCP. Readers interested in the exact details of the simulation set-ups are referred to the files “test-sack” and “sack.tcl” in ns, LBNL's network simulator [MF95]. Similarly, readers interested in the details of the SACK implementations are referred to the simulator code.

The graphs in this section were generated by tracing data 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 90. Packets are numbered starting with packet 0. Each packet arrival and departure is marked by a dot on the graph. For example, a single packet passing through R1 experiencing no appreciable queueing delay would generate 2 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 delay. Packets dropped due to buffer overflow are indicated by an “X” on the graph for each packet dropped.

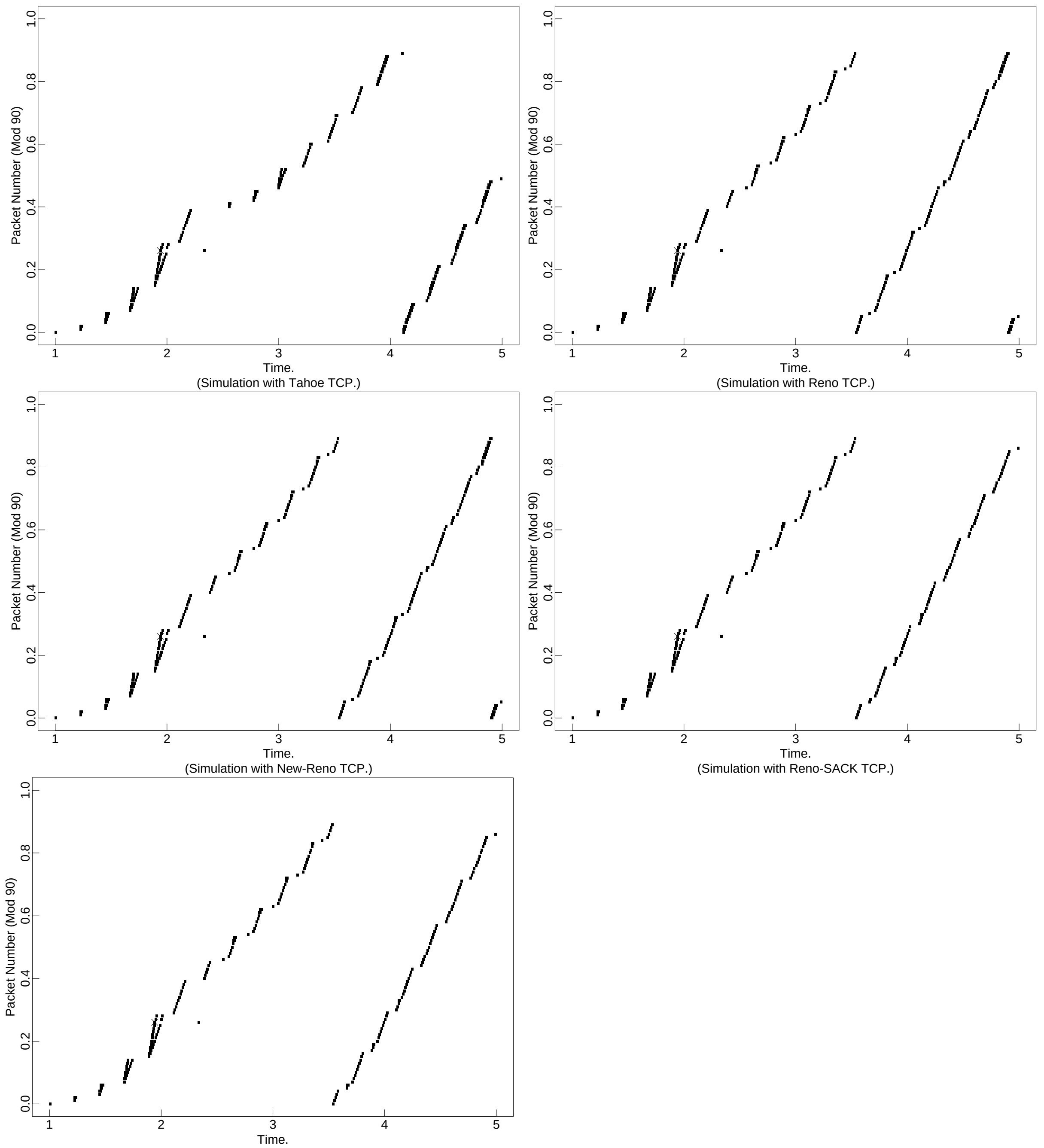

Figure 2: Simulation with one packet loss.

The simulations in Figure 2 show the effect of a single packet drop on Tahoe, Reno, New-Reno, and Reno-SACK TCP. The TCP sender's window size is 14 packets, and the gateway's queue limit is 6 packets. (Note: this is not intended to be a realistic value for a buffer size. This is simply intended as a simple scenario to illustrate the TCP congestion control algorithms.)

For all of the TCP implementations, the dropped packet is detected by the Fast Retransmit procedure, after the source receives three duplicate acknowledgements. None of the sources have to wait for a retransmit timer. After a packet is dropped, the Tahoe TCP slow-starts, while none of the Reno variants slow-start.

The simulations in Figures 3, 4, and 5 were produced simply by changing the window size to 15, 20, and 26 packets, respectively.

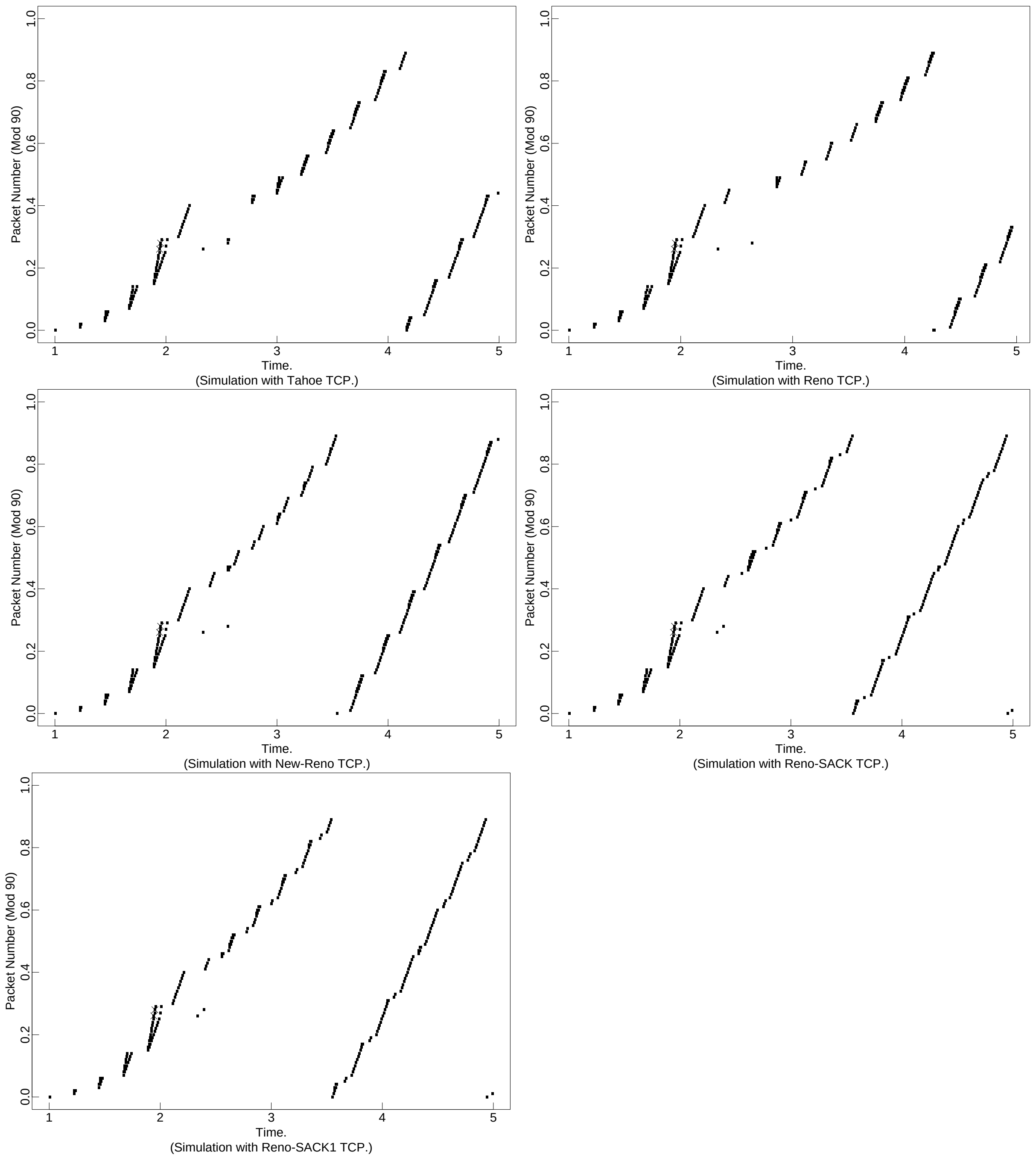

Figure 3: Simulation with two packet losses.

The simulations in Figure 3 show two packets dropped from a single window of data. The Tahoe TCP slow-starts, while all Reno derivatives instead execute the fast recovery algorithm, effectively reducing their congestion window by one half after detecting the dropped packets. In none of these simulations does the source have to wait for a retransmit timer to expire to resend.

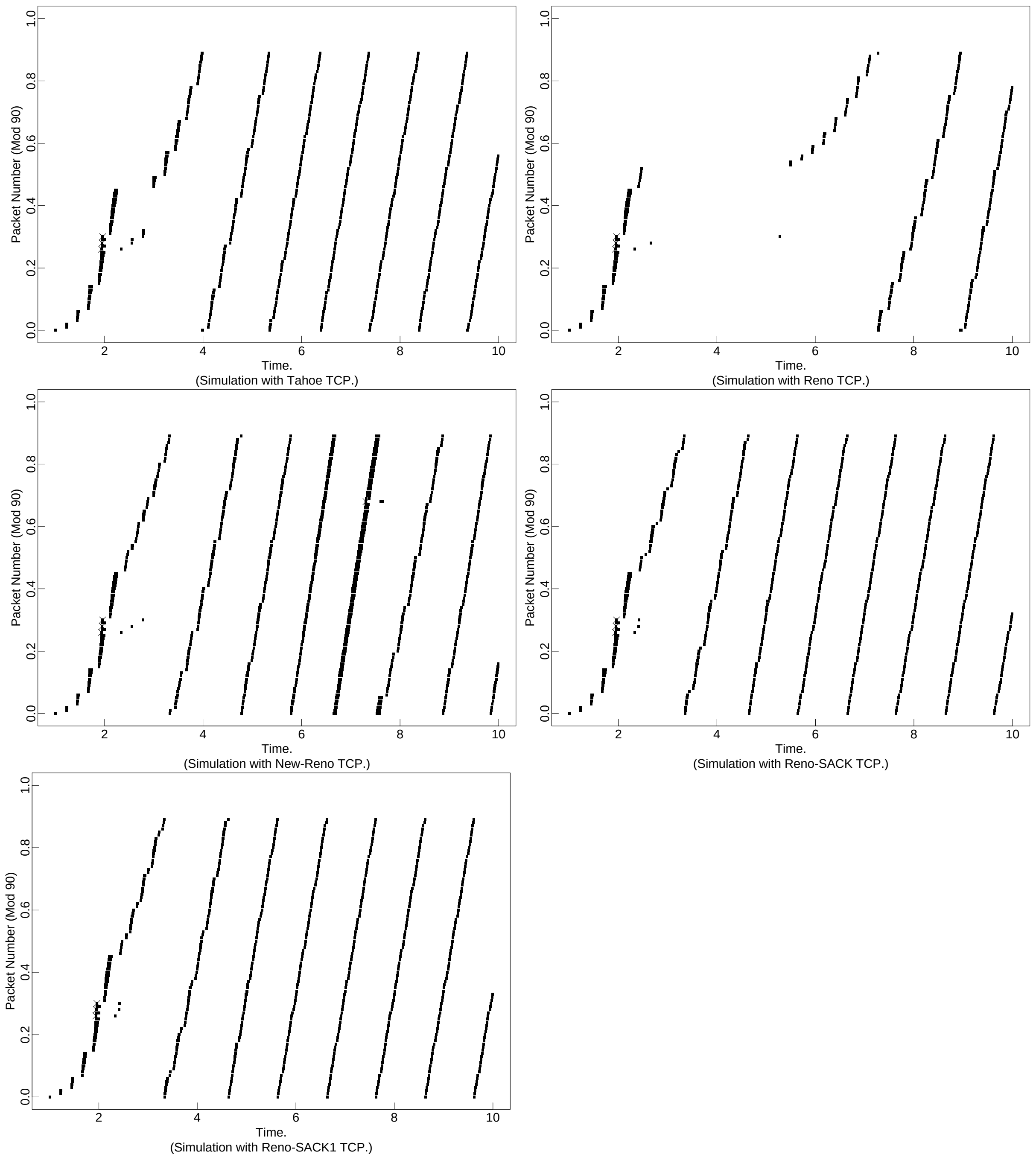

Figure 4: Simulation with three packet losses.

The simulations in Figure 4 show three packets (packets 26, 28, and 30) dropped from a single window of data. These simulations illustrate the differences between Reno and New-Reno TCP. The Reno TCP source is forced to wait for a retransmit timer to expire, while the Tahoe, New-Reno, and Reno-SACK implementations are not.

Here is the description for the Reno source: The Reno source receives three dup acks acknowledging packets up to (and including) packet 25, retransmits packet 26, and enters fast recovery. In the absence of additional information the Reno source makes the optimistic assumption that only one packet in the window has been dropped. When additional dup acks are received, the Reno source transmits at most half a congestion window of new packets.

When the ack for packet 26 is received, acknowledging everything up to packet 27, the Reno source exits fast recovery, but does not retransmit packet 28. However, when three dup acks are received, the Reno source again enters fast recovery, and retransmits packet 28.

The Reno source next receives an acknowledgement for packet 28, acknowledging everything up to packet 29, exits fast recovery, but does not transmit any additional packets. The congestion window does not allow the source to transmit any new packets, and the Reno source has not yet determined that packet 30 has been dropped.

In contrast, the New-Reno source treats “partial acks” received during fast recovery as a special case. These “partial acks” include both the ack for packet 26 and the ack for packet 28. When such a “partial ack” is received, the New-Reno source takes this as an indication of a lost packet, retransmits the packet after the last acked packet, and remains in fast recovery.

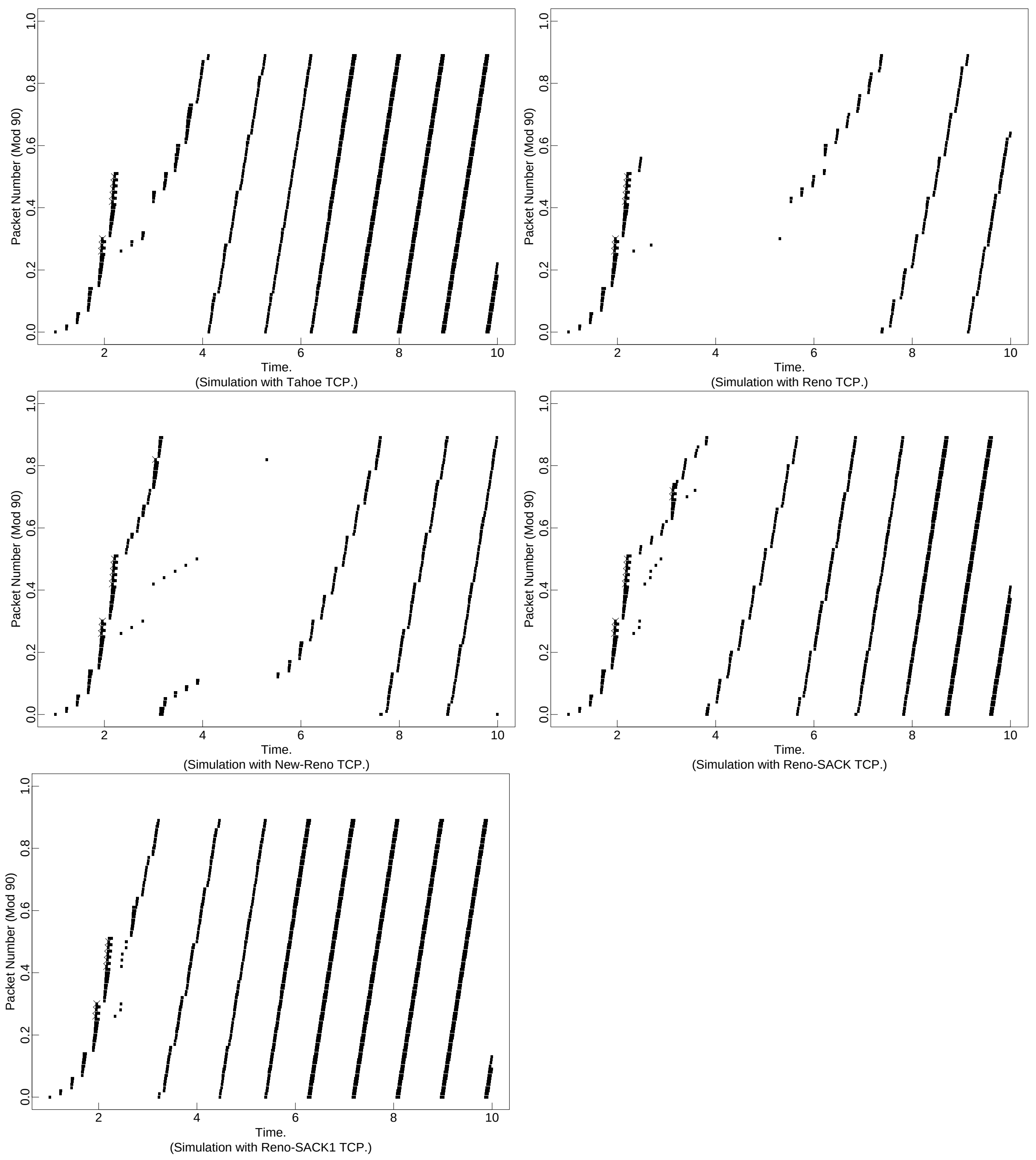

Figure 5: Simulation with multiple packet losses.

The simulations in Figure 5 show multiple packet losses in a window of data. In this case both Reno and New-Reno perform quite badly.

Note that the Reno-SACK source only finds out about three SACK blocks per roundtrip time, and has difficulties keeping the pipe full. The Reno-SACK1 source performs better. First, the Reno-SACK1 source finds out sooner exactly which packets have and have not been dropped, Second, the Reno-SACK1 source uses a slightly different algorithm than the Reno-SACK source for responding to ACK packets that advance the acknowledgement field, but that do not acknowledge all of the packets that were outstanding when Fast Recovery was initiated. The Reno-SACK source retransmits only one packet when such an acknowledgement is received during Fast Recovery. The Reno-SACK source retransmits two packets, thus ensuring that the Reno-SACK source will never do worse that it would if it had used slow-start.

7 Traces

The traces in this section illustrate the poor performance of current implementations of Reno-without-SACK when multiple packets are dropped from one window of data.

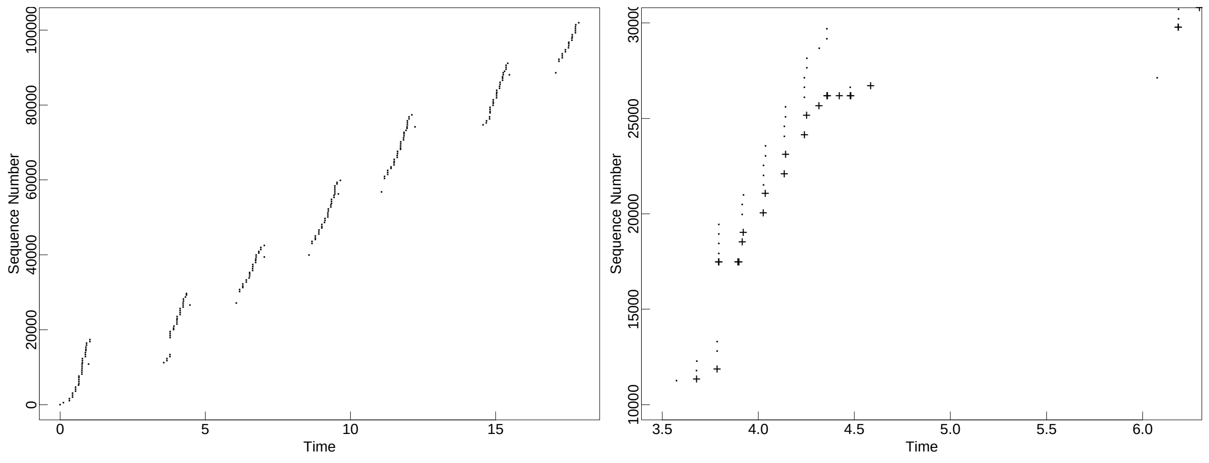

Figure 6: A trace of Reno TCP.

The left figure in Figure 6 shows a trace of a TCP connection from the San Diego Supercomputer Center (running IRIX-5.2) in San Diego to Brookhaven National Laboratory (IRIX-5.1.1) on Long Island. There is a mark for each packet transmitted. The trace clearly shows that the TCP sender repeatedly has to wait for a retransmit timer.

The right figure in Figure 6 is a blow-up of a section of the left figure. In the right figure, there is a dot for every packet transmitted, and a "+" mark for every ACK received. At time 4.5 the sender executes fast retransmit, retransmits one packet, receives an ACK for that packet, and then has to wait for a retransmit timer to expire to retransmit the second packet that was dropped from the original window of data. This is the same as the Reno behavior illustrated in the simulator in Figure 4.

This trace is from 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, a fast retransmit followed by a retransmit timeout, with the additional condition that the packet retransmitted after the retransmit timeout had not been previously retransmitted by the TCP sender. This additional condition is to eliminate Tahoe or Reno traces where the retransmit timeout is simply the result of a retransmitted packet that was itself dropped.)

8 Acknowledgements

Thanks to Vern Paxson for the TCP traces. The implementation of Reno-SACK1 in the simulator is largely from Matt Mathis and Jamshid Mahdavi.

A More simulation results

This section shows some additional simulations comparing Tahoe, Reno, and Reno-SACK TCP.

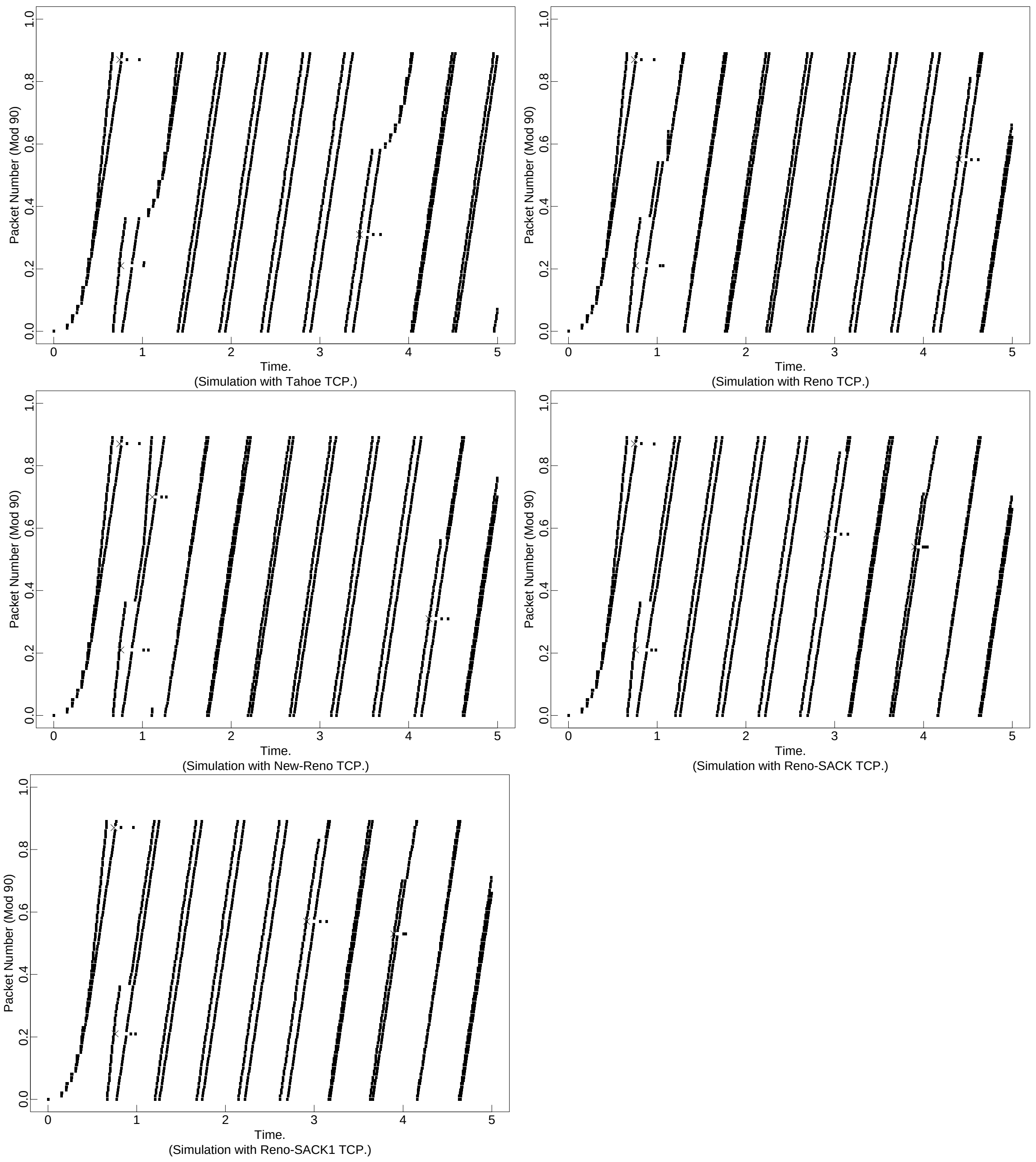

Figure 7: Simulation with delayed-ack sinks, two packets lost from one window of data.

The simulations in Figure 7 show a single TCP connection with two packets dropped from a single window of data.

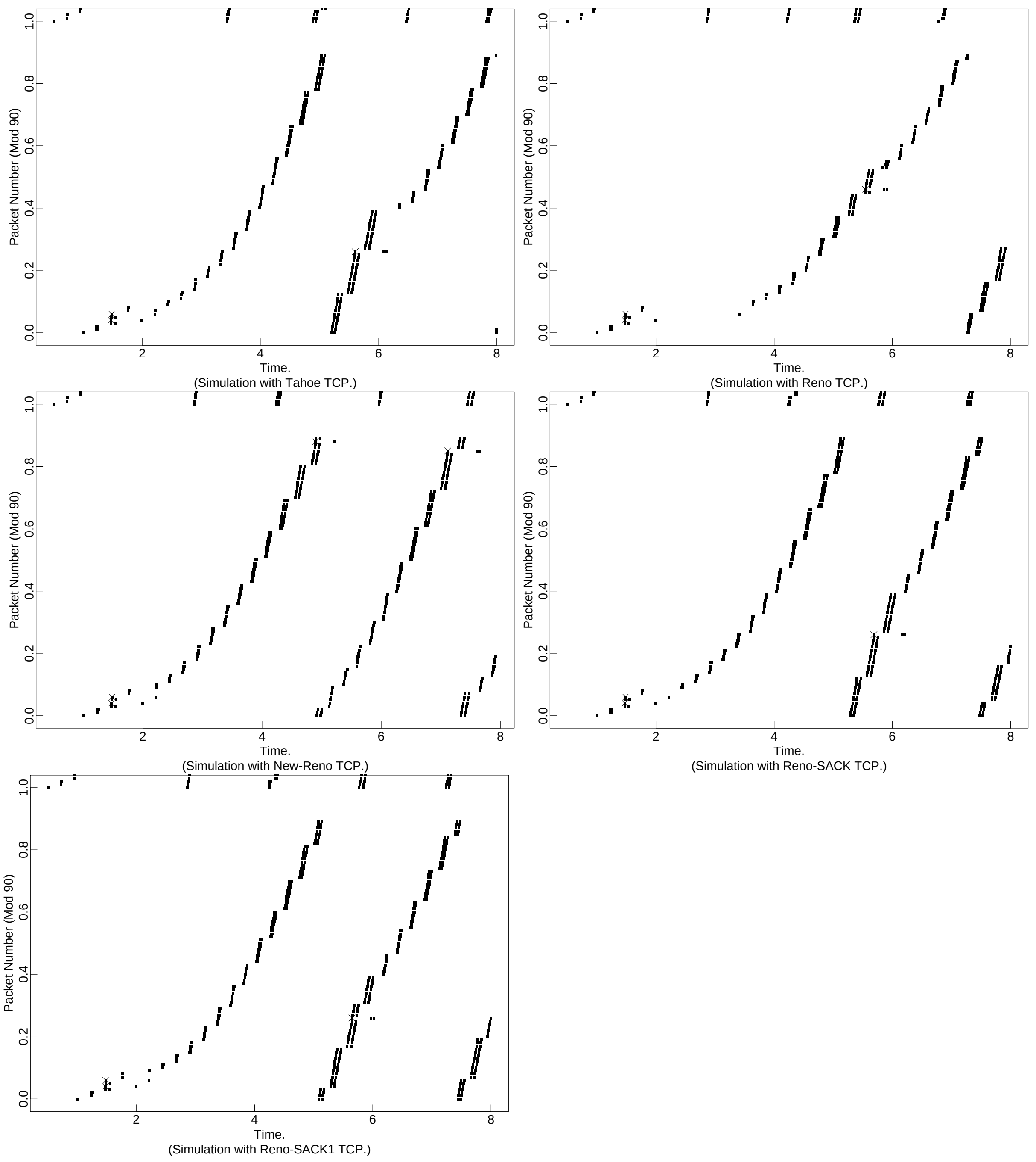

Figure 8: Simulation with two one-way connections, where the connection illustrated has two packets dropped from a small window of data.

The simulations in Figure 8 show a connection with two packets dropped from a single window of data. This and the following simulations show one connection from a simulation with two active TCP connections.

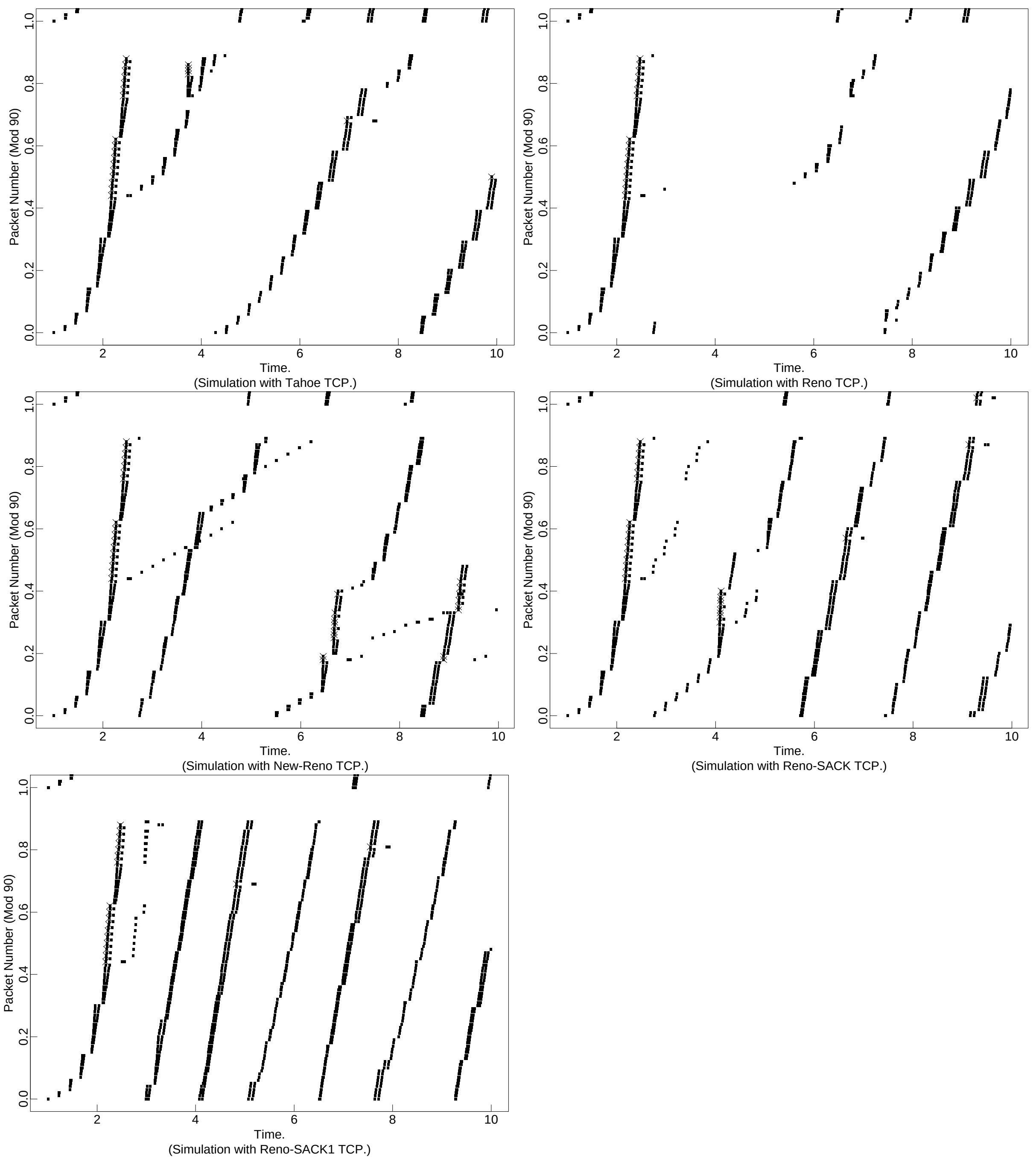

Figure 9: Simulation with two one-way connections, where both connections have multiple packets dropped.

The simulations in Figure 9 show multiple packets dropped from one window of data. The Tahoe simulation at time 4 shows the problems that can result when multiple fast retransmits are performed from packets dropped in a single window of data [Flo94]. This can be corrected in the simulator by setting the TCP option “bug-fix” to “true”.

The New-Reno implementation can send bursts when the sender first comes out of fast recovery. In our simulator these bursts are limited by the parameter “maxburst”, which controls the number of packets that can be sent clocked by a single received ack. For these simulations maxburst is set to four, resulting in a relatively bursty implementation.

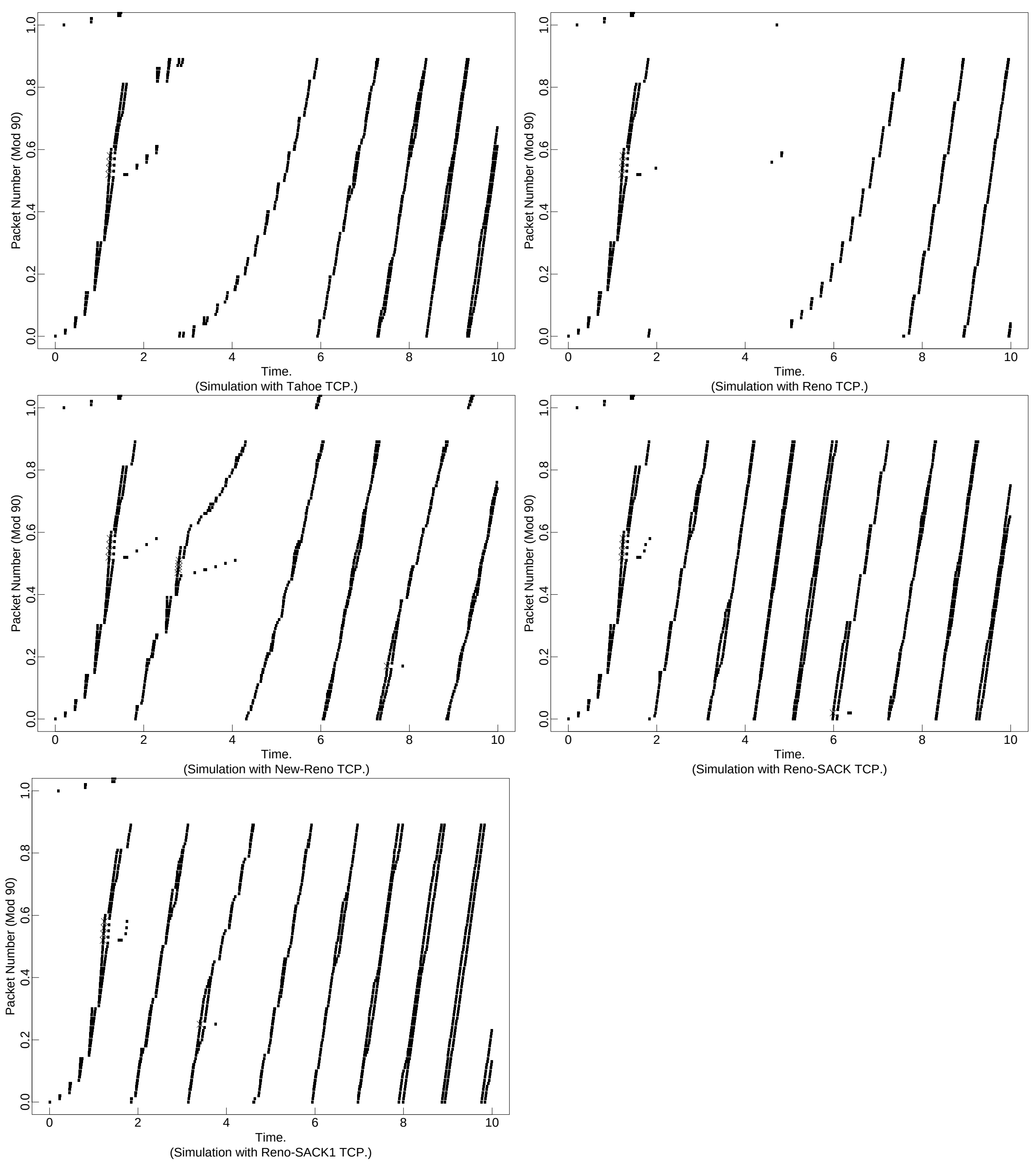

Figure 10: Another simulation with two one-way connections.

The simulations in Figure 10 show multiple packets dropped from one window of data. As in the previous figure, the Tahoe simulation at time 2 shows the problems that can result when multiple fast retransmits are performed from packets dropped in a single window of data [Flo94].

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).

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

[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. Available via http://www-nrg.ee.lbl.gov/nrg-papers.html.

[Flo91] Sally Floyd. "Connections with Multiple Congested Gateways in Packet-Switched Networks Part 1: One-way Traffic,". *ACM Computer Communication Review*, 21(5):30–47, Oct 1991.

[Flo94] Sally Floyd. "TCP and Successive Fast Retransmits,". Technical report, 1994. Available via ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[Flo95] Sally Floyd. "Simulator Tests,". *Unpublished draft*, Jul 1995. Available via http://www-nrg.ee.lbl.gov/nrg-papers.html.

[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).

[Jac88a] 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.

[Jac88b] Van Jacobson. "Dynamic Congestion Avoidance / Control (long message),". *Message to tcp-ip mailing list*, 11 Feb 1988. Available via http://www-nrg.ee.lbl.gov/nrg-email.html.

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

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

[MMFR95] Matthew Mathis, Jamshid Mahdavi, Sally Floyd, and Allyn Romanow. "TCP Selective Acknowledgement Option,". (Internet draft, work in progress), 1995.

[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.