Simulator Tests

Sally Floyd*

Lawrence Berkeley Laboratory
One Cyclotron Road, Berkeley, CA 94704
floyd@ee.lbl.gov

May 7, 1997

1 Introduction

This note shows some of the tests that I use to verify that our simulator is performing the way that we intend it to perform. I have nearly a hundred short tests that I use to verify the simulator; I run these tests after every major change to the simulator. This note shows a selection of these tests. Some of the tests would be of little interest to others, because they check features of the simulator such as the implementation of RED gateways, class-based queueing, variants of TCP with modifications to the window increase algorithms, or non-standard sources.

I first used these tests with the old version of our simulator (tcpsim), and am now (July 1995) using them to validate our new implementation of the simulator (implemented in C++ and in Tcl). (We are gradually making various components from the old simulator available on the new one.) However, I am leaving the input files in this document in the format used by the old simulator.

On each page, the graph shows the results of the simulation. For each graph, the x-axis shows the time in seconds. The y-axis shows the packet number mod 90. There is a mark on the graph for each packet as it arrives and departs from the congested gateway, and a "x" for each packet dropped by the gateway. Some of the graphs show more that one active connection. In this case, packets numbered 1 to 90 on the y-axis belong to the first connection, packets numbered 101 to 190 on the y-axis belong to the second connection, and so on.

Below the graph is the input file for the simulator. The first part of the input file gives the simulator parameters that differ from the default parameters given in the standard input file. The second part of the input file defines the simulation network. The input file shows the edges in the network, with the queue parameters for the output buffers for the forward and/or backward direction for each link (if the output buffer does not use the default queue, which is an unbounded queue). Following the convention in our new simulator, the output buffer size includes a buffer for the packet currently being transmitted on the output link.

The third part of the file contains a line for each active connection, specifying the application (e.g., ftp, telnet), the transport protocol ("tcp" for Tahoe-style TCP, "renotcp" for Reno-style TCP, and "satcp" for Reno-style TCP with Selective Acknowledgements), and the variant of the transport protocol used by the receiver ("sink" for a TCP receiver that sends an ACK packet for every data packet, "dasink" for a TCP receiver that sends delayed ACKs, and "sasink" for a TCP receiver that sends Selective ACKs). The delayed-ACK receiver delays sending an ACK until a second data packet arrives, or until 100 msec.

Below the input file is a brief description of the TCP behavior demonstrated in the test. Included are the commands for running this test on our old simulator *tcpsim* (which is not publically available), and on our new network simulator *ns*.

2 Simulator defaults, disclaimers, and assumptions

This simulator is not intended to reproduce the behavior of specific implementations of TCP, and we do not use production TCP code in our simulator. The simulator is intended to explore the behavior inherent to the underlying congestion control algorithms, including the Slow Start, Congestion Avoidance, Fast Retransmit, and Fast Recovery algorithms. Even with the extensive testing that we have done, I would be surprised if the simulator was without bugs. Nevertheless, I have reasonable confidence that the simulations with this simulator display the essential dynamics of TCP's congestion control algorithms.

Several aspects of the simulator don't match the behavior of actual implementations at all. For example, in the simulator each TCP connection deals with packets, not segments. For each connection, it is possible to specify the data and acknowledgement packet sizes in bytes, but the simulator does not provide for two-way data within a single con-

*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. This is an expanded version of a note that was first made available in October 1994.

nection. Thus, there is no provision for acknowledgements piggy-backed on data packets.

Several parts of the behavior of the simulations in this note do not match current TCP implementations. For example, except for the few tests where a delayed ACK receiver policy is specified, the TCP receiver acks every packet.

For these simulations, the granularity of the TCP clock is set to 100 msec. This means that roundtrip times are measured only to the nearest 100 msec.

3 General warnings

The two simulations with phase effects are intended partly to emphasize the dangers of interpreting an individual simulation. The results of individual simulations might be sensitive to the exact parameters used in the simulation, such as propagation delays, TCP window sizes, etc. My experience is that simulations are more reliably used to show how performance varies as a function of a particular parameter (such as propagation delay, TCP window size, number of connections, number of congested gateways, etc.).

4 Acknowledgements

Our old simulator tcpsim is a version of the REAL simulator [K88a] built on Columbia's Nest simulation package [BDSY88a], with extensive modifications and bug fixes made by Steven McCanne and by Sugih Jamin. For the new simulator ns [Ns], this has been rewritten embedded into Tcl, with the simulation engine implemented in C++.

References

[Ns] Ns. Available via http://www-nrg.ee.lbl.gov/ns/.

[BDSY88a] Bacon, D,. Dupuy, A., Schwartz, J., and Yemimi, Y., "Nest: a Network Simulation and Prototyping Tool", *Proceedings of Winter 1988 USENIX Conference*, 1988, pp. 17-78.

[F96] Floyd, S., "Simulator Tests for Random Early Detection (RED) Gateways", URL "ftp://ftp.ee.lbl.gov/papers/redsims.ps.Z", October 1996.

[F94] Floyd, S., "TCP and Successive Fast Retransmits", technical report, October 1994. URL ftp://ftp.ee.lbl.gov/papers/fastretrans.ps.

[F94a] Floyd, S., "TCP and Explicit Congestion Notification", *ACM Computer Communication Review*, V. 24 N. 5, October 1994, p. 10-23. Available via http://www-nrg.ee.lbl.gov/nrg/.

[FJ92] Floyd, S., and Jacobson, V., "On Traffic Phase Effects in Packet-Switched Gateways", *Internetworking: Research and Experience*, V.3 N.3, September 1992, pp. 115-156. Available via http://www-nrg.ee.lbl.gov/nrg/.

[FJ93] Floyd, S., and Jacobson, V., *Random Early Detection Gateways for Congestion Avoidance*, IEEE/ACM Transactions on Networking, V.1 N.4, August 1993, p. 397-413. Available via http://www-nrg.ee.lbl.gov/nrg/.

[K88a] Keshav, S., "REAL: a Network Simulator", *Report 88/472*, Computer Science Department, University of California at Berkeley, Berkeley, California, 1988.

5 Tahoe TCP: Slow Start, Congestion Avoidance, Fast Retransmit

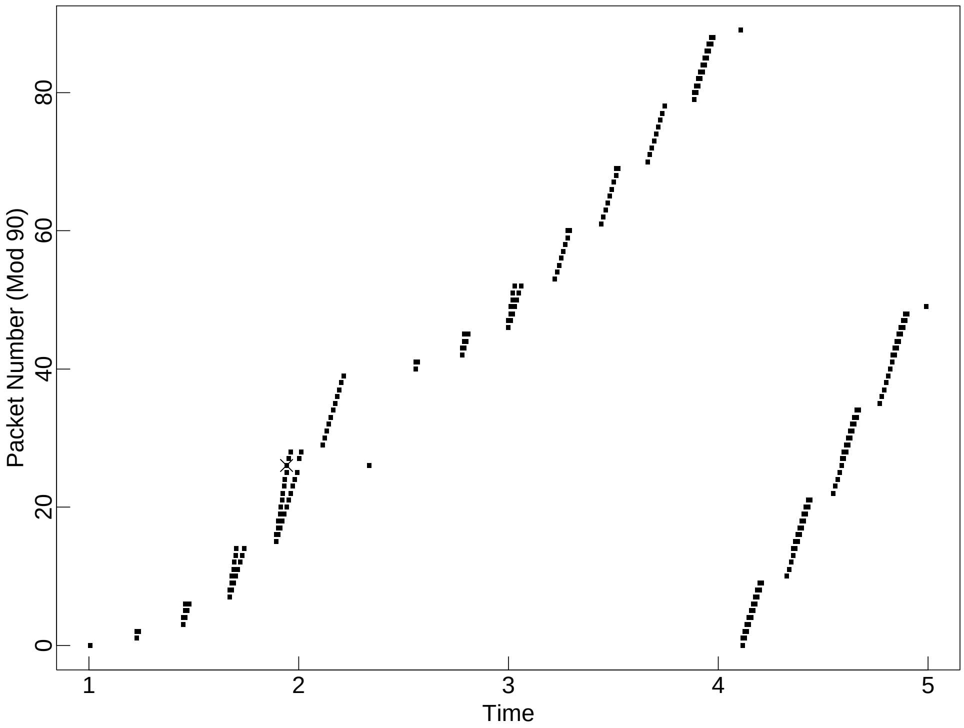

Figure 1: Fast Retransmit, Slow Start, and Congestion Avoidance algorithms, with a single packet drop.

```
renotcp, tcp [ window=14 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
ftp conv from tcp [start-at=1.0] at s1 to sink at k1
```

This test shows the Fast Retransmit, Slow Start, and Congestion Avoidance algorithms of Tahoe TCP. Initially, the connection increases its window using Slow-Start. The sender detects a dropped packet after receiving three duplicate ACKs; this is the Fast Retransmit algorithm. The sender invokes Slow-Start, and later increases the window using the Congestion Avoidance algorithm.

In this simulation, the congestion window is 14 packets when the first packet is dropped. After the Fast Retransmit, the source slow-starts up to a congestion window of 7 packets, and then opens the congestion window further by roughly one packet per roundtrip time.

This test is run on ns with “ns test-suite.tcl tahoe2”, and on tcpsim with “csh test14C.com”. Figure 14 shows the Fast Recovery algorithm for this scenario.

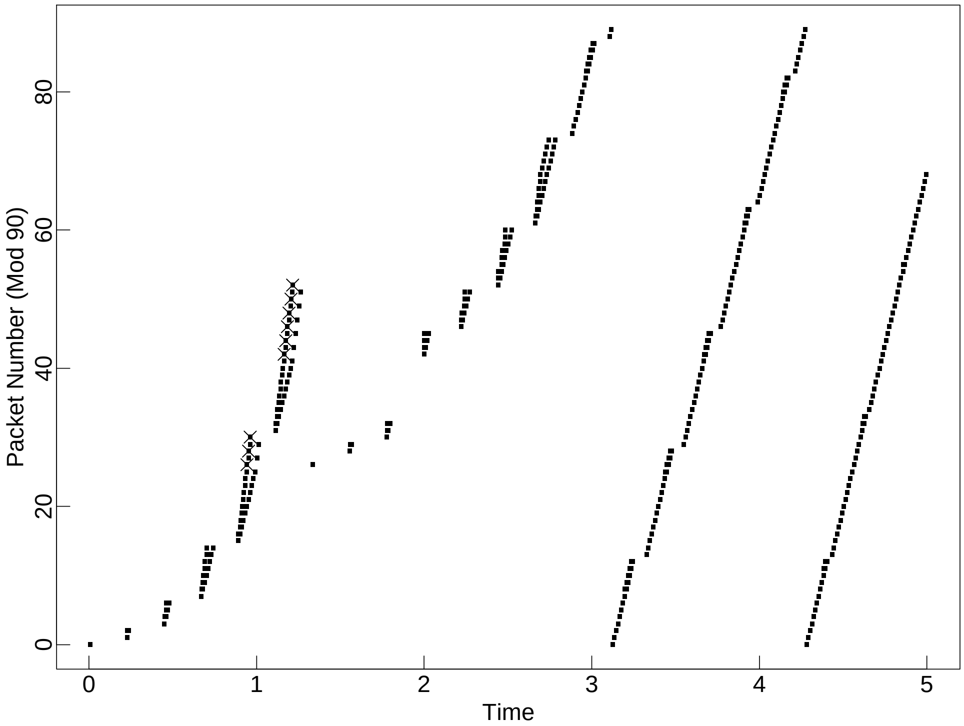

Figure 2: Fast Retransmit, Slow Start, and Congestion Avoidance algorithms, with multiple packet drops.

```
tcp [ window=50 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
        forward [ queue-size=6 ]
ftp conv from tcp at s1 to sink at k1
```

This test shows the Fast Retransmit, Slow Start, and Congestion Avoidance algorithms of Tahoe TCP with multiple packet drops for one window of data.

This test is run on ns with “ns test-suite.tcl tahoe1”, and on tcpsim with “csh test1.com”.

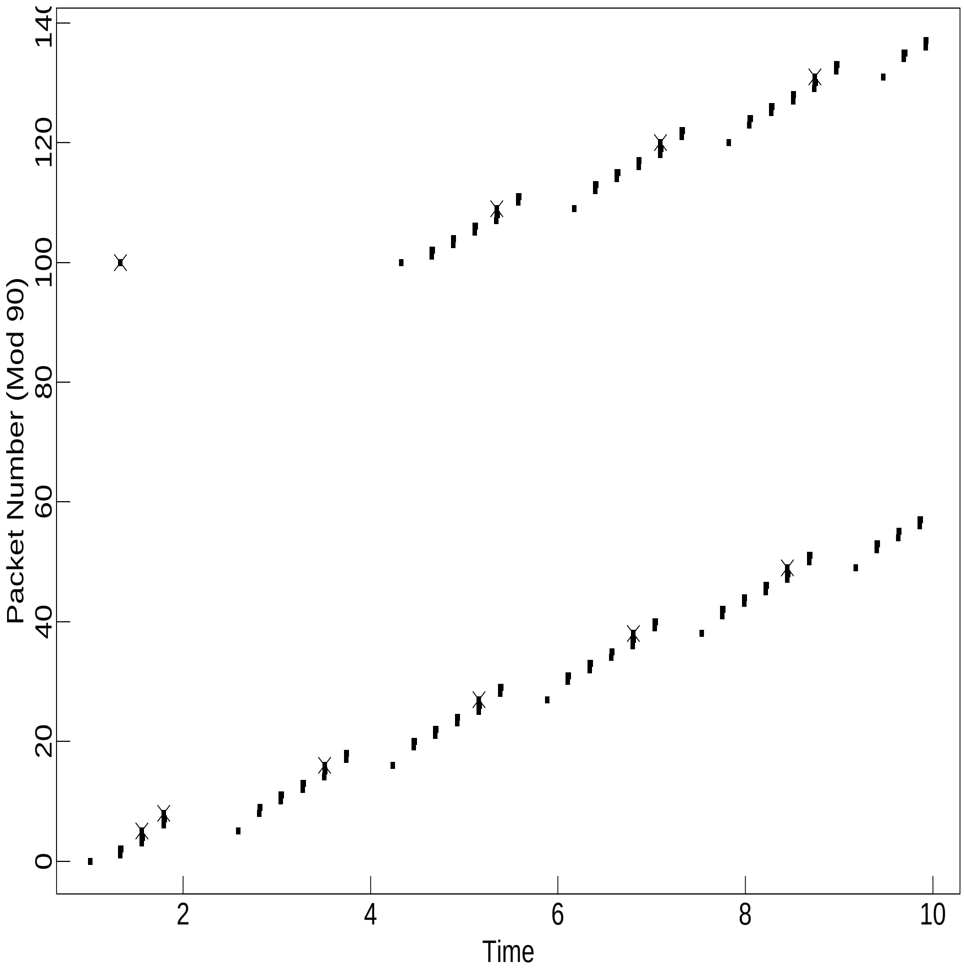

Figure 3: Retransmit timers.

```
renotcp, tcp [ window=4 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=2 ]
ftp conv from tcp [start-at=1.0] at s1 to dasink at k1
ftp conv from tcp [start-at=1.3225] at s2 to dasink at k1
```

This test shows two connections each with a maximum window of four packets. Because of the small window coupled with the use of delayed acks, the source will never receive three duplicate ACKs after a packet drop, and will always have to recover from packet drops by waiting for a retransmit timer.

For the top connection, the first packet of the connection is dropped. This shows the default value for the retransmit timer, which in this simulation is three seconds. For the bottom connection, a packet is dropped only after several successful measurements of the roundtrip time have been made. The tcp implementation in this simulation uses the timestamp option, where the roundtrip time can be measured for every new ACK packet.

This test is run on ns with “ns test-suite.tcl timers”, and on tcpsim with “csh test3.com”.

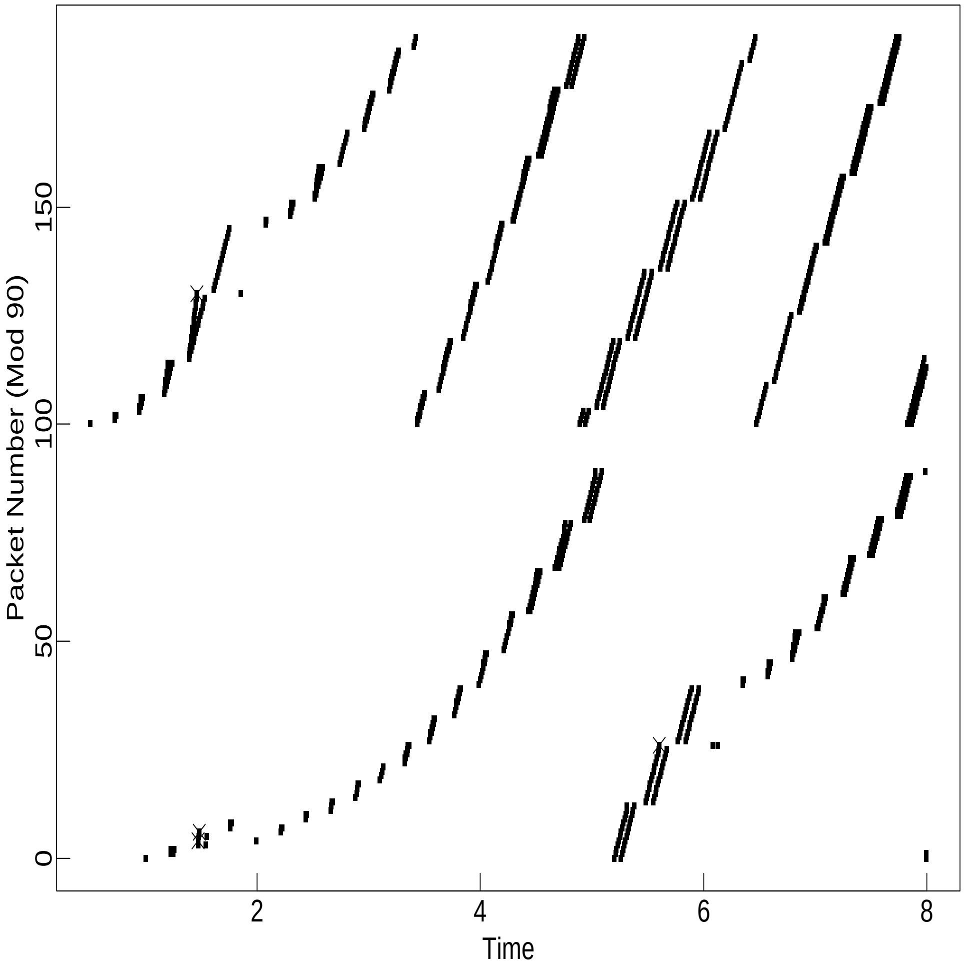

Figure 4: Fast Retransmit, Slow Start, and Congestion Avoidance algorithms, with multiple packet drops.

```
renotcp [ window=100 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=8 ]
ftp conv from tcp [start-at=1.0] at s1 to sink at k1
ftp conv from tcp [start-at=0.5 window=16] at s2 to sink at k1
```

This test shows Tahoe TCP with two packet drops from one window of packets. Figure 18 shows this scenario with Reno TCP, and Figure 21 shows Reno TCP with Selective Acknowledgements.

This test is run on ns with “ns test-suite.tcl tahoe3”, and on tcpsim with “csh test15C.com”.

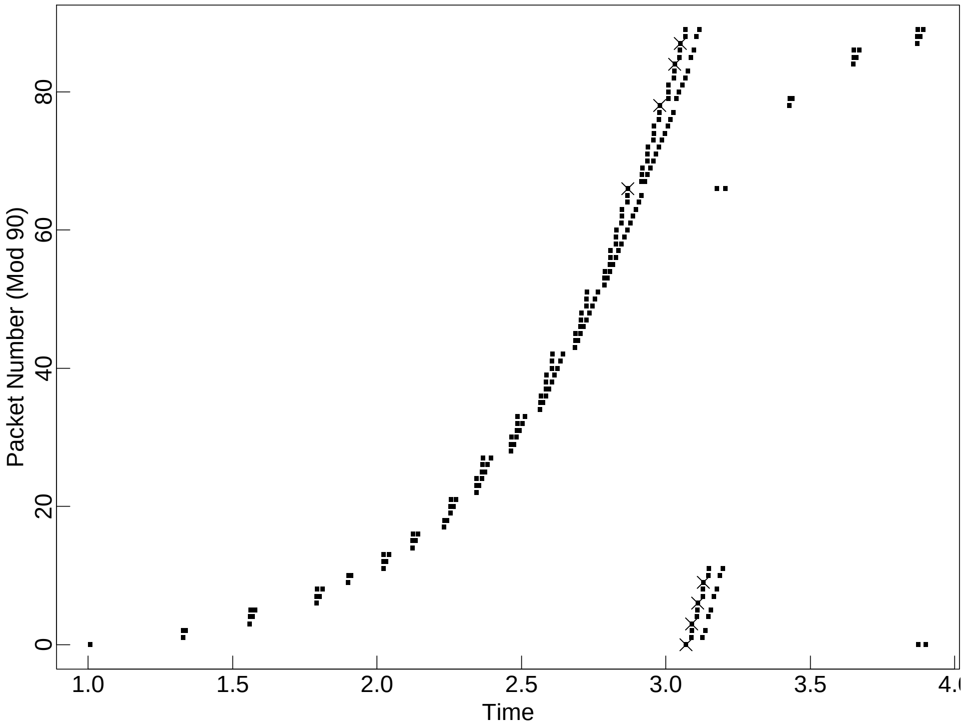

Figure 5: Delayed-ACK sink.

```
renotcp, tcp [ window=50 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
ftp conv from tcp [start-at=1.0] at s1 to dasink at k1
```

This test shows the delayed-ACK sink. The ACK for the first packet is delayed by 100 msec. The second and third packets are acknowledged by a single ACK, and therefore the window is increased by only one packet, from two to three. When the single ACK arrives for packets 4 and 5, the window is again increased to four. The source gets to send three packets. When the delayed ACK arrives for packet 6, the window is increased again to five, allowing the source to send two more packets.

This test is run on ns with "ns test-suite.tcl delayed", and on tcpsim with "csh test7.com".

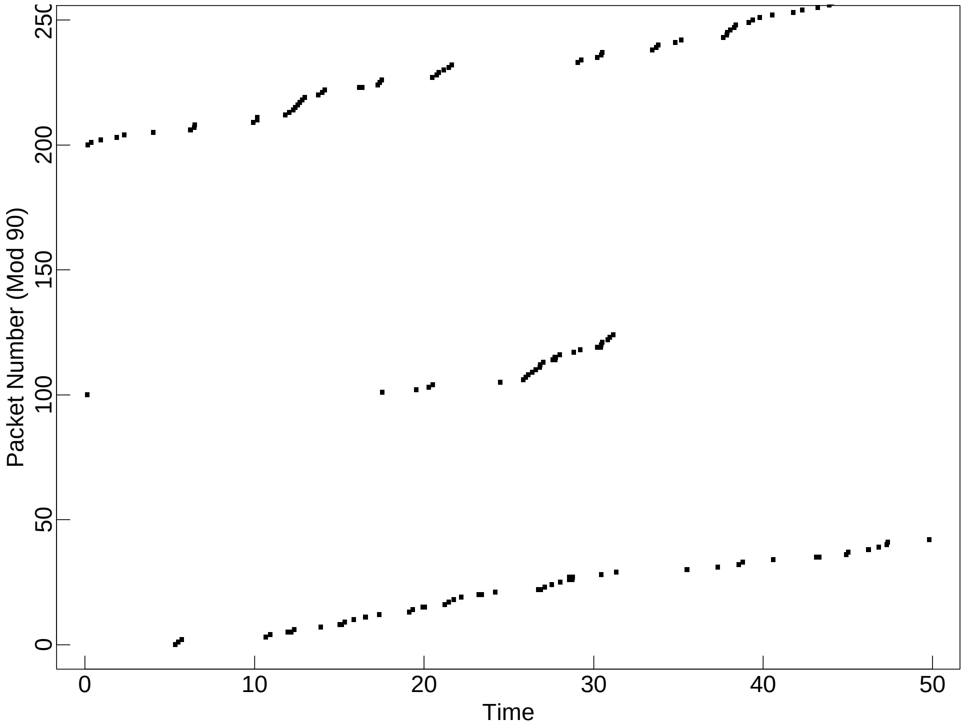

Figure 6: Telnet connections.

```
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
telnet [interval=1100ms] conv from tcp at s1 to sink at k1
telnet [interval=0] conv from tcp at s2 to sink at k1
telnet [interval=0] conv from tcp at s2 to sink at k1
```

This test shows various telnet sources. The first telnet connection generates fixed-size packets with interpacket times from an exponential distribution with a mean of 1.1 seconds. The second and third telnet connections generate packets with interpacket times from the tcplib distribution. This test will give quite different results for different seeds for the pseudo-random number generator.

This test is run on ns with “ns test-suite.tcl telnet”, and on tcpsim with “csh test10.com”.

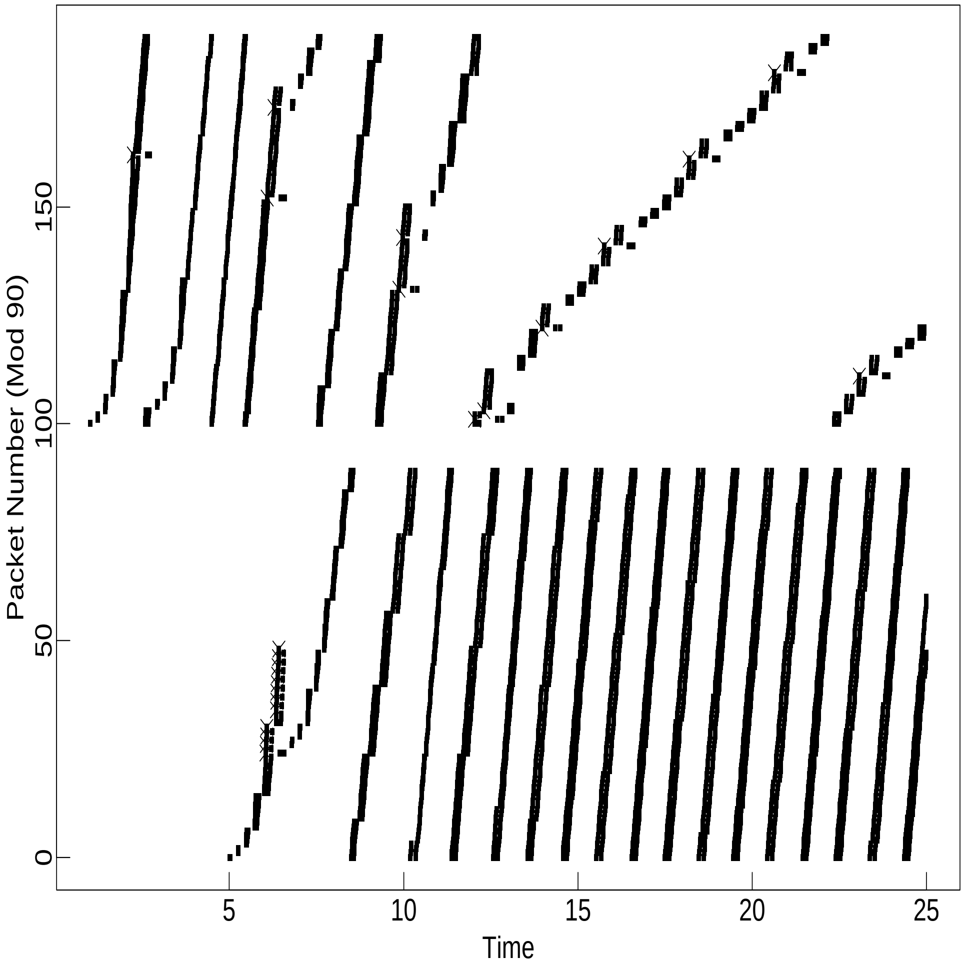

Figure 7: Phase effects.

```
renotcp, tcp [ window=32 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 3ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=16 ]
ftp conv from tcp [start-at=5.0] at s1 to sink at k1
ftp conv from tcp [start-at=1.0] at s2 to sink at k1
```

This test shows a simulation of two TCP connections, with slightly different propagation delays on the two incoming edges. Note that the top connection, from source s2, receives a disproportionate share of the packet drops. This is due to phase effects [FJ92]. These phase effects remain with Reno TCP sources and with delayed-ACK sinks.

This test is run on ns with “ns test-suite.tcl phase”, and on tcpsim with “csh test20.com”.

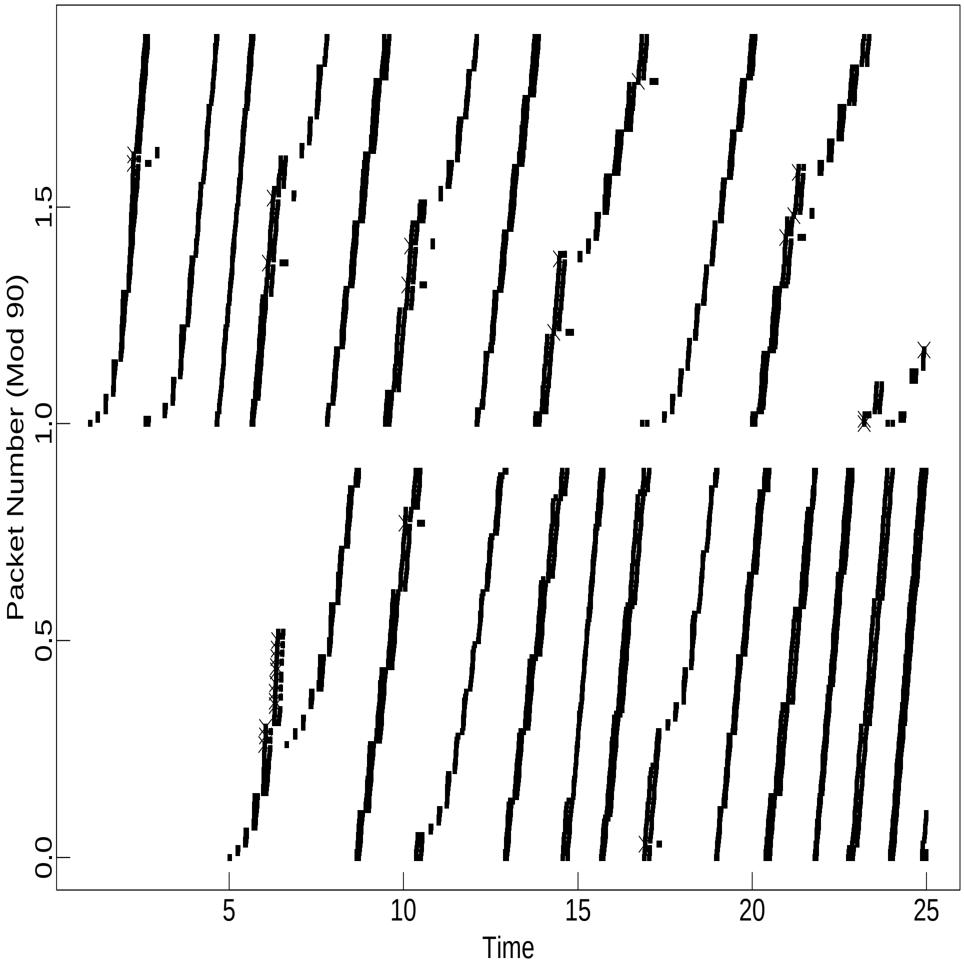

Figure 8: Phase effects, with random overhead.

```
renotcp, tcp [ window=32 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 3ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=16 ]
ftp conv from tcp [start-at=5.0 overhead=10ms] at s1 to sink at k1
ftp conv from tcp [start-at=1.0 overhead=10ms] at s2 to sink at k1
```

This test shows a simulation of two TCP connections identical to that in Figure 7, except that each TCP connection uses a random overhead of up to 10ms (the transmission delay at the bottleneck gateway) at the source to add a random delay to the processing time for each incoming ACK packet. The purpose of this random delay is to add a sufficient random factor to the simulations to prevent phase effects [FJ92].

This test is run on ns with "ns test-suite.tcl phase2".

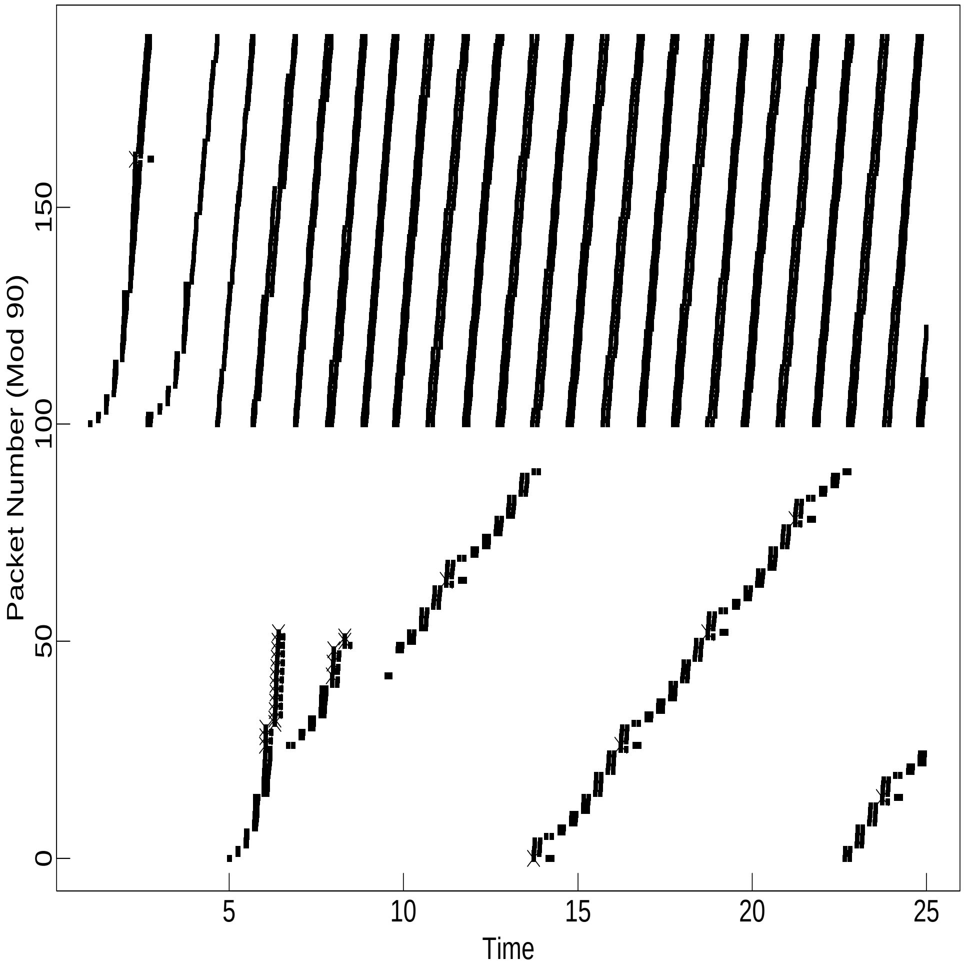

Figure 9: Phase effects simulation, with changed propagation delays.

```
renotcp, tcp [ window=32 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 9.5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=16 ]
ftp conv from tcp [start-at=5.0] at s1 to sink at k1
ftp conv from tcp [start-at=1.0] at s2 to sink at k1
```

This test shows a simulation of two TCP connections, with a slightly different propagation delay on one of the links, compared to the test in Figure 7. Note that the bottom connection receives a disproportionate share of the packet drops.

This test is run on ns with “ns test-suite.tcl phase1”, and on tcpsim with “csh test24.com”.

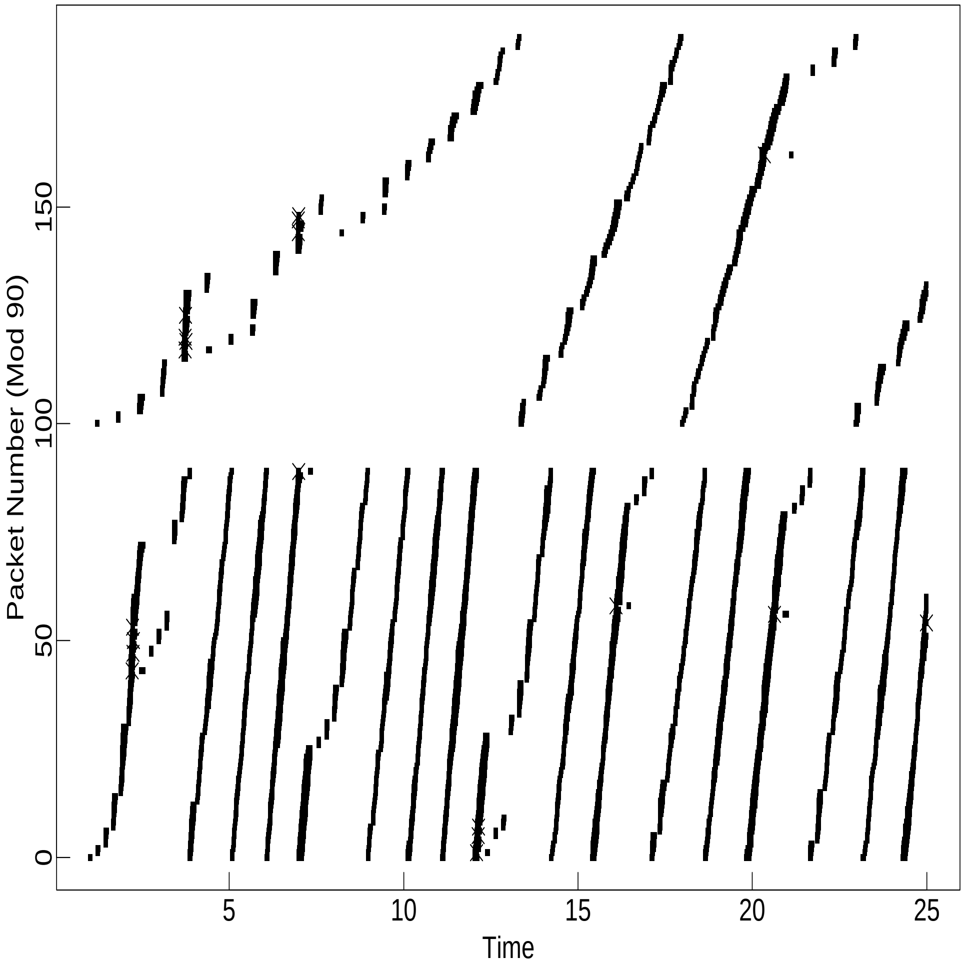

Figure 10: Connections with different roundtrip times.

```
tcp [  window=30 ]
bq [ dropmech=random-drop ]
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=11  ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 200ms
ftp conv from tcp at s1 to sink at k1
ftp conv from tcp at s2 to sink at k1
```

This test shows two Tahoe TCP connections where top connection (from source s2) has a roundtrip time roughly three times that of the bottom connection. The shared gateway uses Random Drop. Because of TCP's window increase algorithms, the connection with the shorter roundtrip time increases its window faster, and receives a disproportionate share of the link bandwidth

This test is run on ns with “ns test-suite.tcl tahoe4”, and on tcpsim with “csh test32.com”.

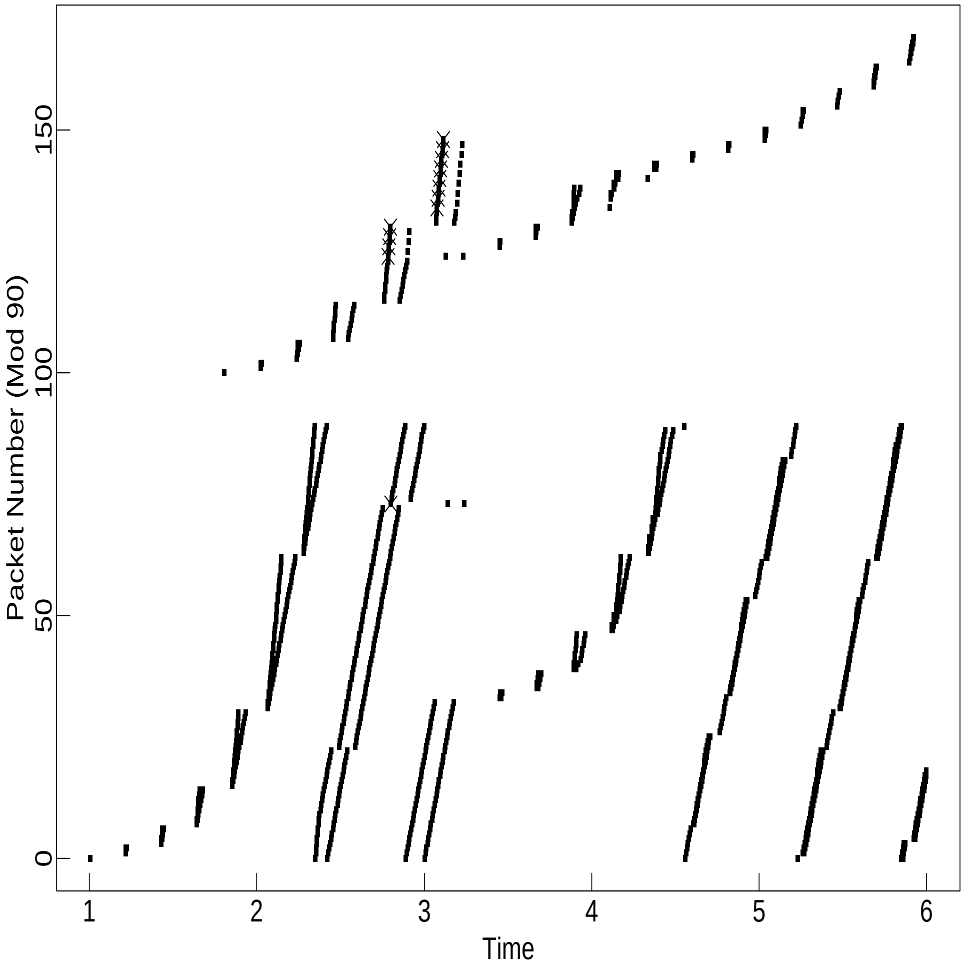

Figure 11: Multiple fast retransmits.

```
tcp [  window=50 ]
edge r1 to k1 bandwidth 1.5Mb  delay 100ms
    forward [ queue-size=23  ]
edge s1 to r1 bandwidth 10Mb delay 3ms
edge s2 to r1 bandwidth 10Mb delay 5ms
ftp conv from tcp [start-at=1.0] at s1 to sink at k1
ftp conv from tcp [start-at=1.8 stop-at-packet=100] at s2 to sink at k1
```

This test shows the pathological behavior of Tahoe TCP when there are multiple fast retransmits that result from losses in one window of data. By varying the start time of the second TCP connection, it is possible to completely change the nature of the pathological behavior.

This test is run on ns with “ns test-suite.tcl bug”, and on tcpsim with “csh test100.com”. Figure 12 shows the same scenario with the simple fix to the TCP code that is described in [F94].

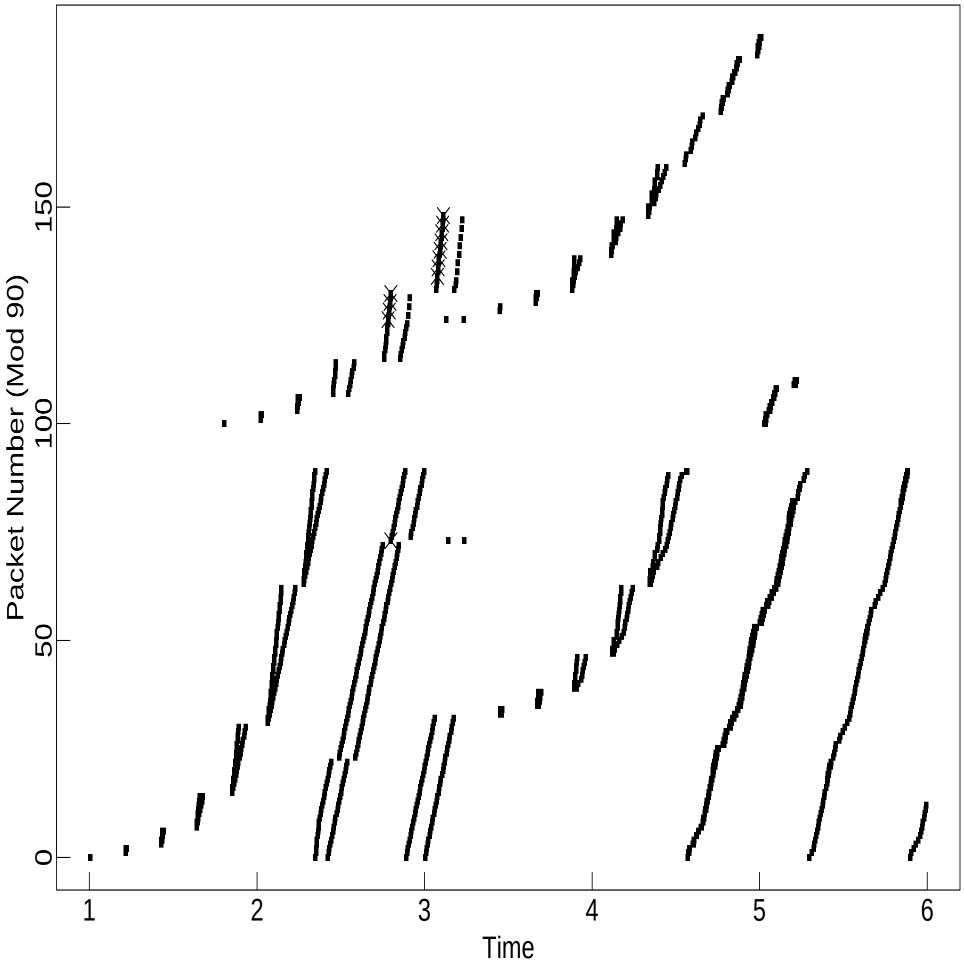

Figure 12: TCP with a bug-fix that prevents multiple fast retransmits.

This test is run on ns with “ns test-suite.tcl no_bug”, and on tcpsim with “csh test100.com”.

6 Reno TCP: Fast Recovery

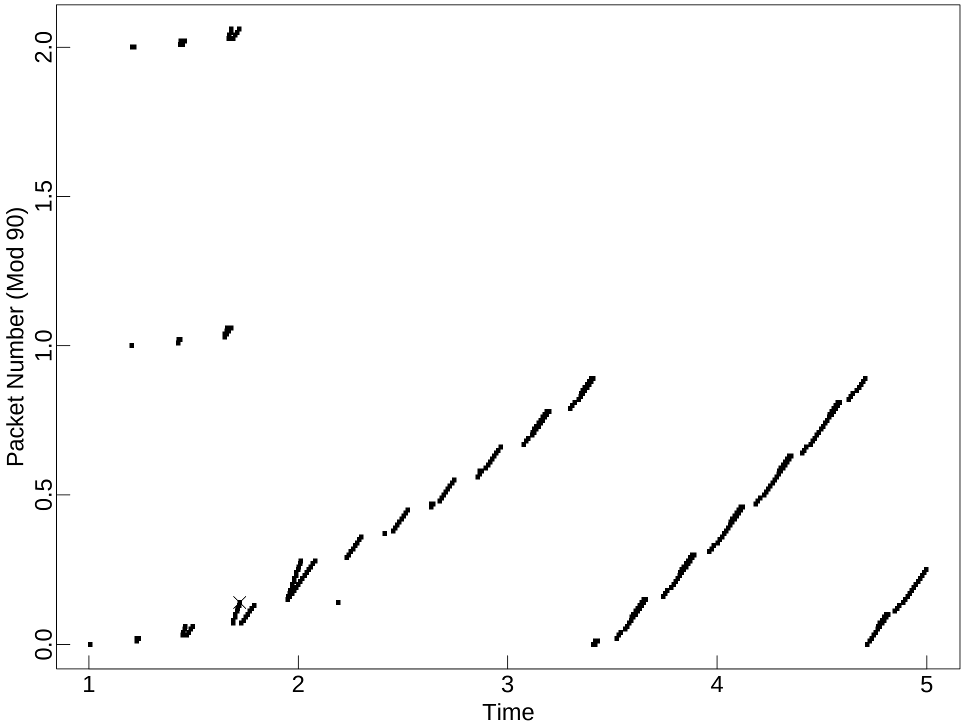

Figure 13: Fast Recovery, with a single packet drop.

```
renotcp, tcp [ window=28 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=8 ]
ftp conv from renotcp [start-at=1.0] at s1 to sink at k1
ftp conv [maxpkts=7] from renotcp [start-at=1.2] at s1 to sink at k1
ftp conv [maxpkts=7] from renotcp [start-at=1.2] at s1 to sink at k1
```

This test shows the Fast Recovery algorithm with a single packet drop. The bottom row shows the packets from the first connection, the middle row shows the packets from the second connection, and the top row shows the packets from the third connection. The second and third connections are added only to get the desired packet drop pattern from the bottom connection.

This test is run on ns with “ns test-suite.tcl renoA”.

7 Reno TCP: Fast Recovery

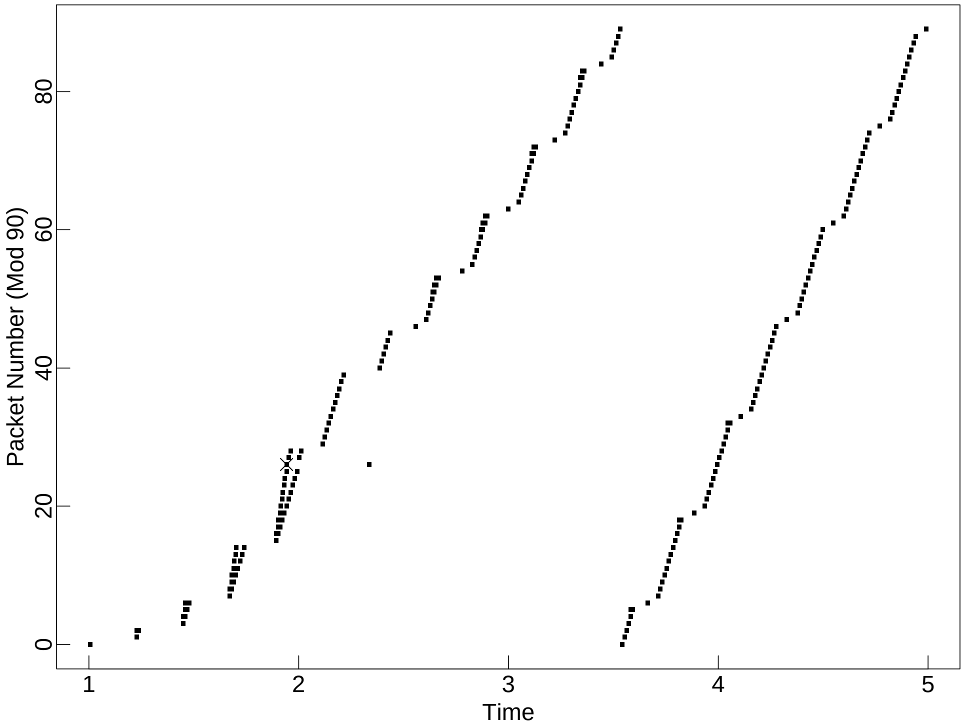

Figure 14: Fast Recovery, with a single packet drop, and different values for the receiver's advertised window and the maximum congestion window.

```
renotcp, tcp [ maxcwnd=14 window=28 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
ftp conv from renotcp [start-at=1.0] at s1 to sink at k1
```

This test shows the Fast Recovery algorithm with a single packet drop. Figure 14 and Figure 1 differ in that Figure 14 uses Reno-style TCP while Figure 1 uses Tahoe-style TCP.

In order to show easily simulate a range of scenarios, and to get the best possible behavior of the Reno congestion control algorithms, this simulation runs with the receiver's advertised window of 28, but with a maximum congestion window of 14. This separation between the receiver's advertised window, intended to prevent the receiver's buffer from overflowing, and the maximum congestion window, intended to limit the number of packets outstanding in the pipe, allows us to easily simulate a range of scenarios, without the sender being unnecessarily limited by the receiver's advertised window during Fast Recovery. In actual TCP implementations using the Reno algorithm, the sender's maximum congestion window is always set to the receiver's advertised window.

This test is run on ns with “ns test-suite.tcl reno”, and tcpsim with “csh test14B.com”. This test with selective acks is run on ns with “ns test-suite-sack.tcl sack1”.

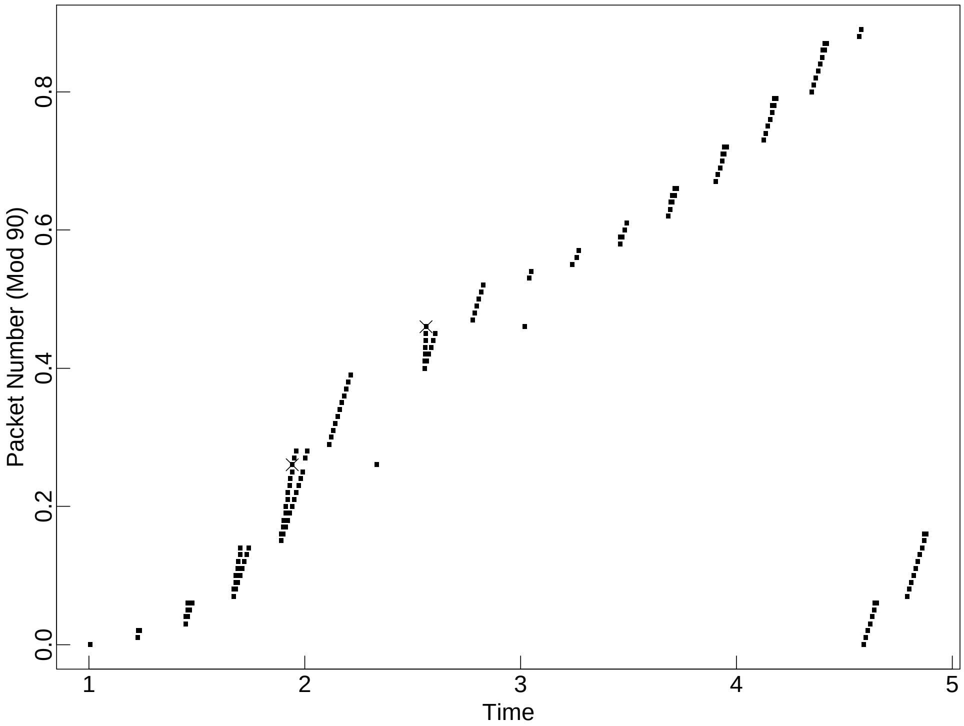

Figure 15: Fast Recovery, with a single packet drop.

```
renotcp, tcp [ maxcwnd=14 window=14 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
ftp conv from renotcp [start-at=1.0] at s1 to sink at k1
```

This test shows the Fast Recovery algorithm with a single packet drop, with both the receiver's advertised window and the maximum congestion window set to 14. Figure 15 and Figure 1 differ only in that Figure 14 uses Reno-style TCP while Figure 1 uses Tahoe-style TCP. Unlike the previous two figures showing Fast Recovery, in this figure the sender is unable to send new packets between the initiation of Fast Retransmit, and the receipt of the acknowledgement for the retransmitted packet, due to the limitations imposed by the receiver's advertised window. Instead, the sender sends an entire window of packets when the acknowledgment for the retransmitted packet arrives.

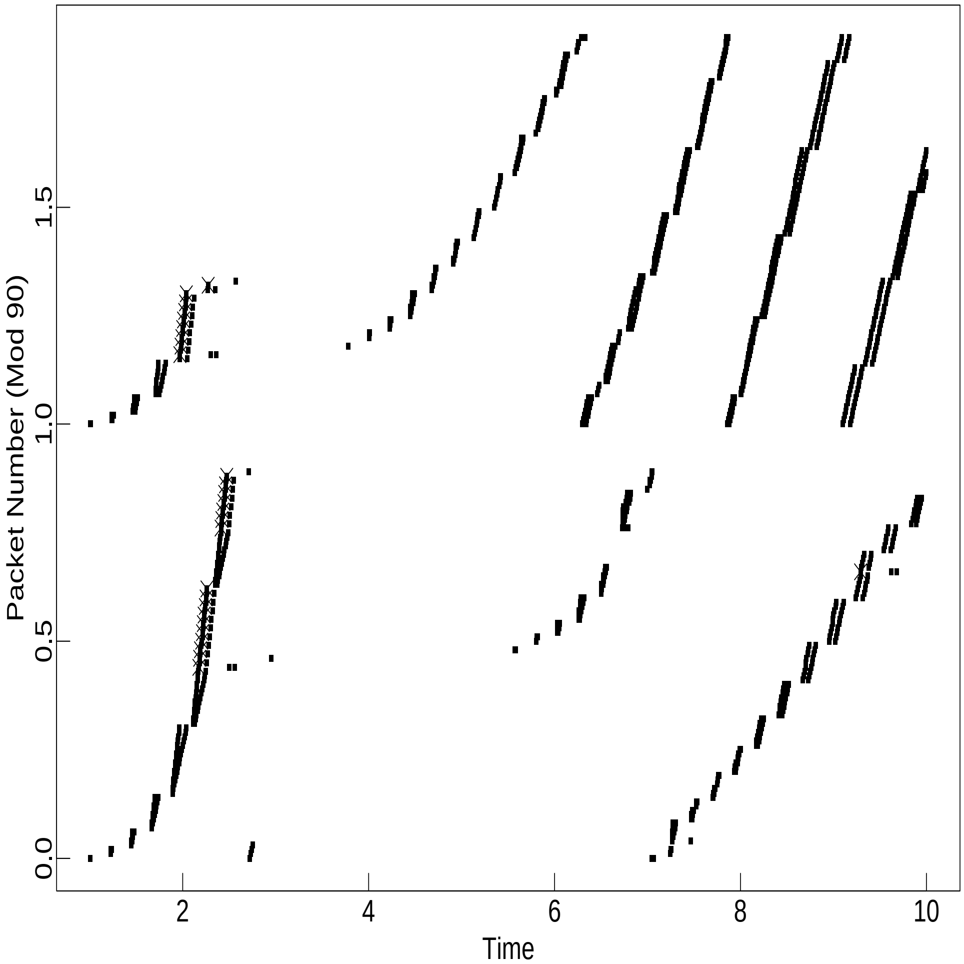

Figure 16: Fast Recovery, with multiple packet drops.

```
renotcp, tcp [ window=50 bug-fix=false]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=9 ]
ftp conv from renotcp [start-at=1.0] at s1 to sink at k1
ftp conv from renotcp [start-at=1.0 window=20] at s2 to sink at k1
```

This test shows the Fast Recovery algorithm in Reno TCP, with multiple packet drops from one window of packets. TCP has to wait for a retransmit timer to recover.

This test is run on ns with "ns test-suite.tcl reno5".

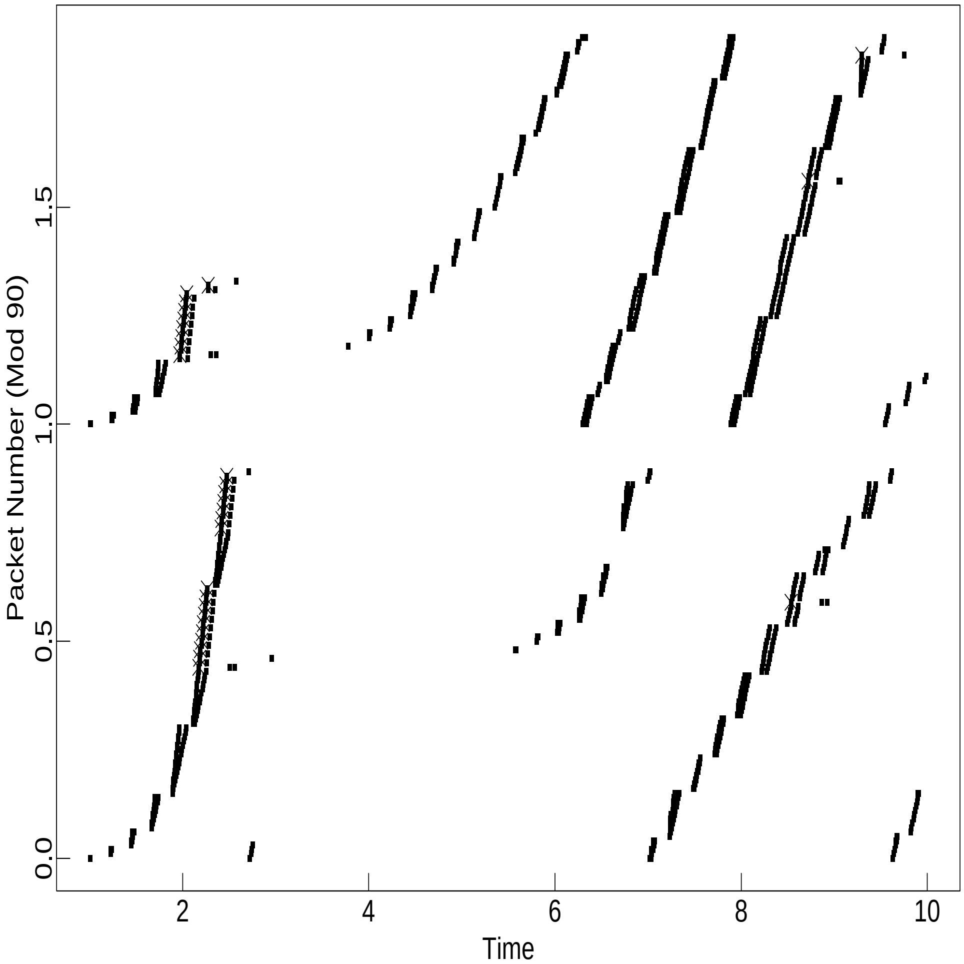

Figure 17: Fast Recovery, with multiple packet drops.

```
renotcp, tcp [ window=50 bug-fix=true ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=9 ]
ftp conv from renotcp [start-at=1.0] at s1 to sink at k1
ftp conv from renotcp [start-at=1.0 window=20] at s2 to sink at k1
```

This test is identical to that in Figure 16, except that the TCP sources are modified to implement a bug-fix that prevents unnecessary fast-retransmits following a retransmit timeout.

This test is run on ns with “ns test-suite.tcl reno2”.

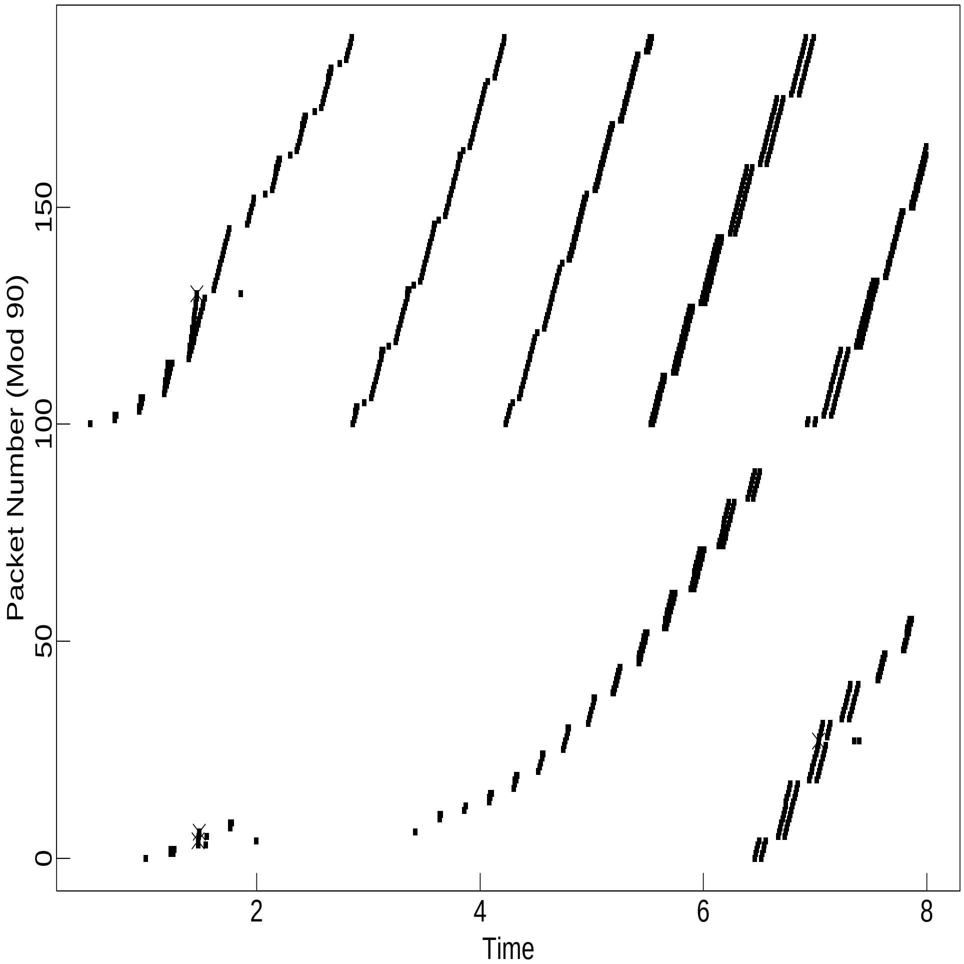

Figure 18: Fast Recovery, with multiple packet drops.

```
renotcp [ window=100 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=8 ]
ftp conv from renotcp [start-at=1.0] at s1 to sink at k1
ftp conv from renotcp [start-at=0.5 window=16] at s2 to sink at k1
```

This test shows the Fast Recovery algorithm in Reno TCP, with multiple packet drops from one window of packets. This test was intentionally designed to highlight the weakness of Reno TCP with multiple packet drops in one window of data. Figure 4 shows Tahoe TCP in the same scenario, and Figure 21 shows TCP with Selective ACKs.

For the Reno TCP connection in the bottom row, packets 5 and 7 are dropped, and the receiver sends three duplicate ACKs when it receives packet 6, 8, and 9. The source receives the three duplicate ACKS, and uses fast retransmit to retransmit packet 5. When the source receives an ACK acknowledging up to and including packet 6, TCP has to wait for a retransmit timer to recover.

This test is run on ns with "ns test-suite.tcl reno3", and on tcpsim with "csh test15B.com".

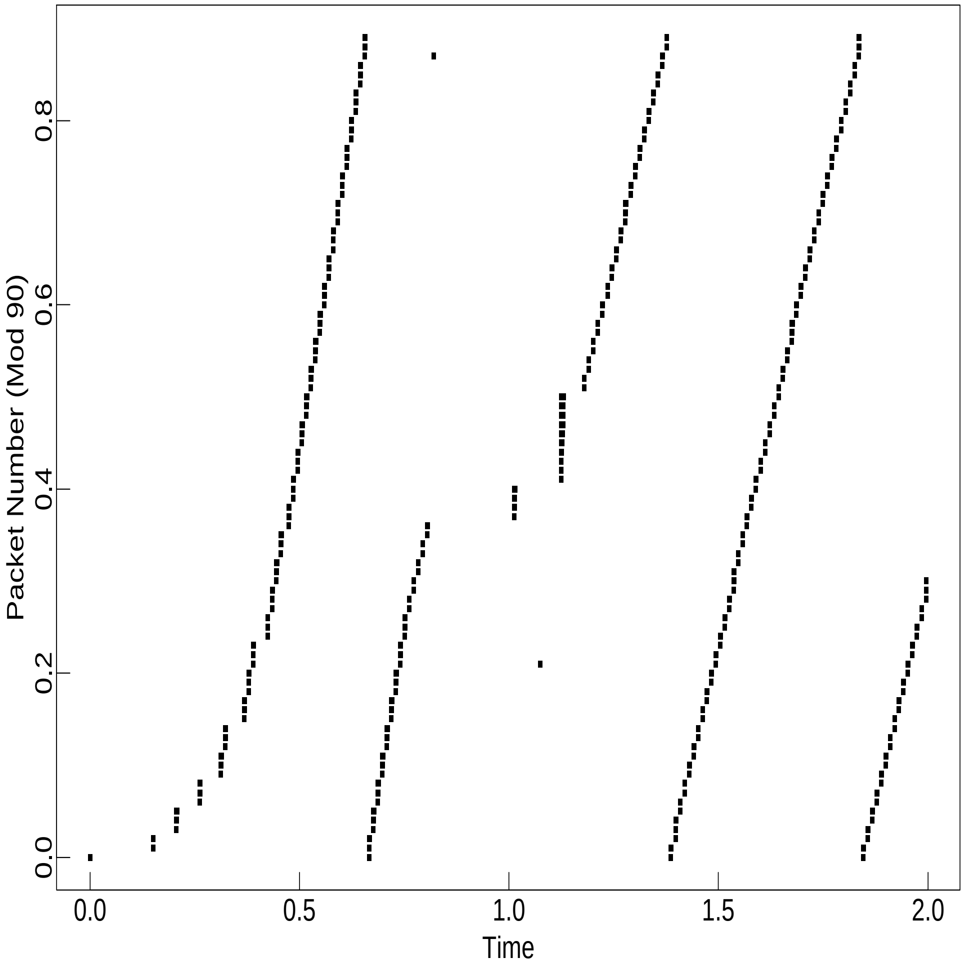

Figure 19: Fast Recovery, with two packet drops.

```
renotcp [ window=40 ]
edge s1 to r1 bandwidth 10Mb delay 2ms
edge r1 to r2 bandwidth 1.5Mb  delay 20ms
    forward [ queue-size=29 ]
ftp conv from renotcp [start-at=1.0] at s1 to dasink at r2
```

This test shows the Fast Recovery algorithm in Reno TCP with two packet drops from one window of packets, in a scenario where the Reno algorithm recovers from the two drops in one window of data without having to wait for a retransmit timer. This test shows that although Reno is sometimes capable of recovering from two packet drops without waiting for a retransmit timer to expire, this recovery is often accompanied by a burst of packets transmitted back-to-back from the source.

Two packets, packets 88 and 112, were dropped on the link from ”r1” to ”r2”. This trace shows packets transmitted on the link from ”s1” to ”r1”, and does not show packet drops for another link. When the acknowledgement for the first dropped packet is received, followed by three dup acks, the source does a second fast retransmit, retransmitting the second dropped packet. When the acknowledgement for the second dropped packet is received, the acknowledgement acknowledges all packets that have been transmitted so far, and the source transmits a window of packets back-to-back.

This test is run on ns with “ns test-suite.tcl reno4a”.

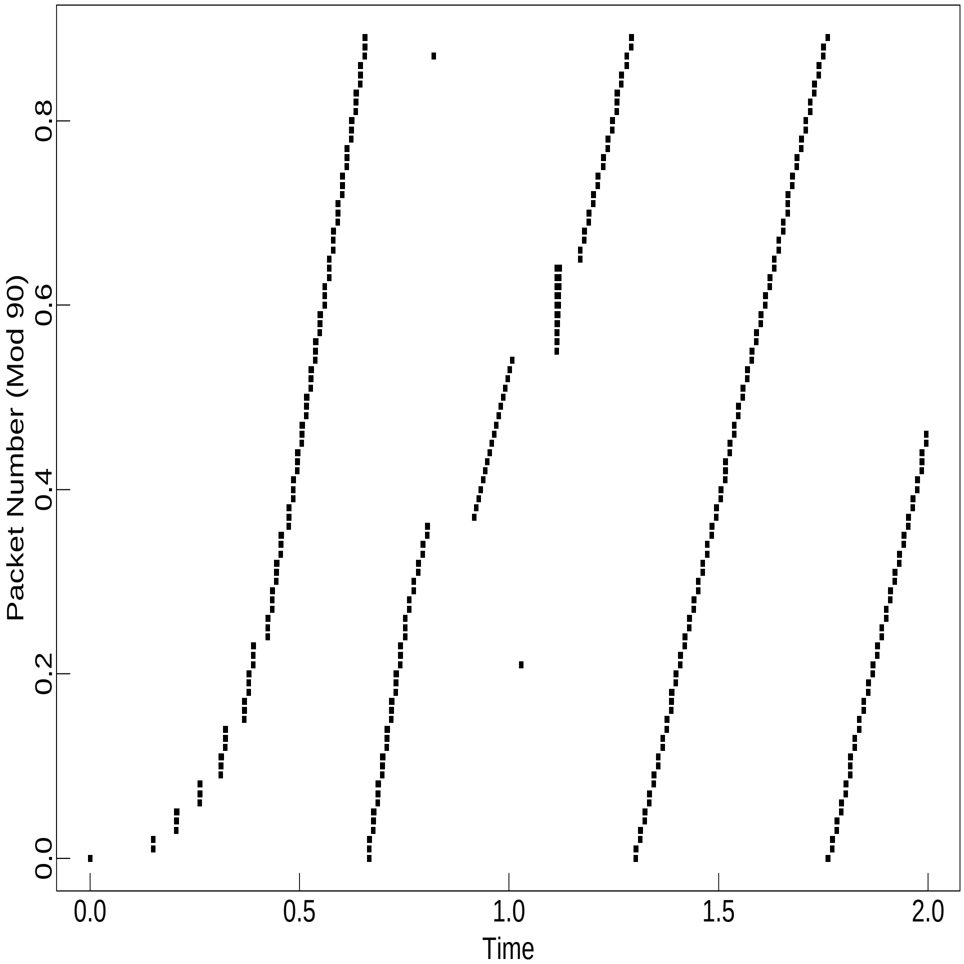

Figure 20: Fast Recovery, with two packet drops, and different values for the receiver's advertised window and the maximum congestion window.

```
renotcp [ maxcwnd=40 window=80 ]
edge s1 to r1 bandwidth 10Mb delay 2ms
edge r1 to r2 bandwidth 1.5Mb  delay 20ms
    forward [ queue-size=29 ]
ftp conv from renotcp [start-at=1.0] at s1 to dasink at r2
```

This test differs from Figure 19 only in that, in this test, the sender limits itself to a maximum congestion window of 40, but assumes a receiver's advertised window of 80. This shows that the sender sends a burst of packets even in a scenario where the sender is not limited by the receiver's advertised window.

This test is run on ns with “ns test-suite.tcl reno4”.

8 Reno TCP with Selective Acknowledgements

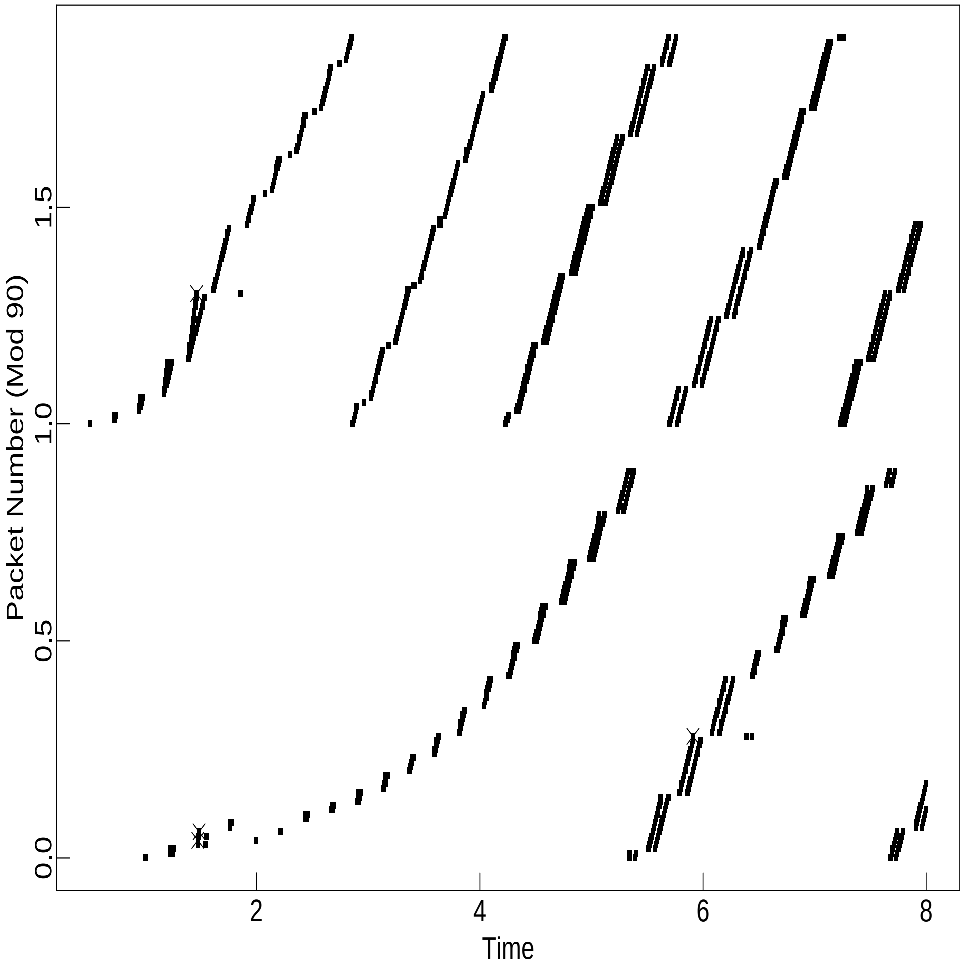

Figure 21: Fast Recovery with Selective Acknowledgements, two packets dropped in a single window.

```
satcp [ window=100 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge s2 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=8 ]
ftp conv from satcp [start-at=1.0] at s1 to sasink at k1
ftp conv from satcp [start-at=0.5 window=16] at s2 to sasink at k1
```

This test shows the Fast Recovery algorithm in TCP with Selective Acknowledgements. This simulation with Tahoe TCP is shown in Figure 4, and the simulation with Reno TCP is shown in Figure 18.

This test is run on tcpsim with “csh test15D.com” and on ns with “ns test-suite-sack.tcl sack3”.

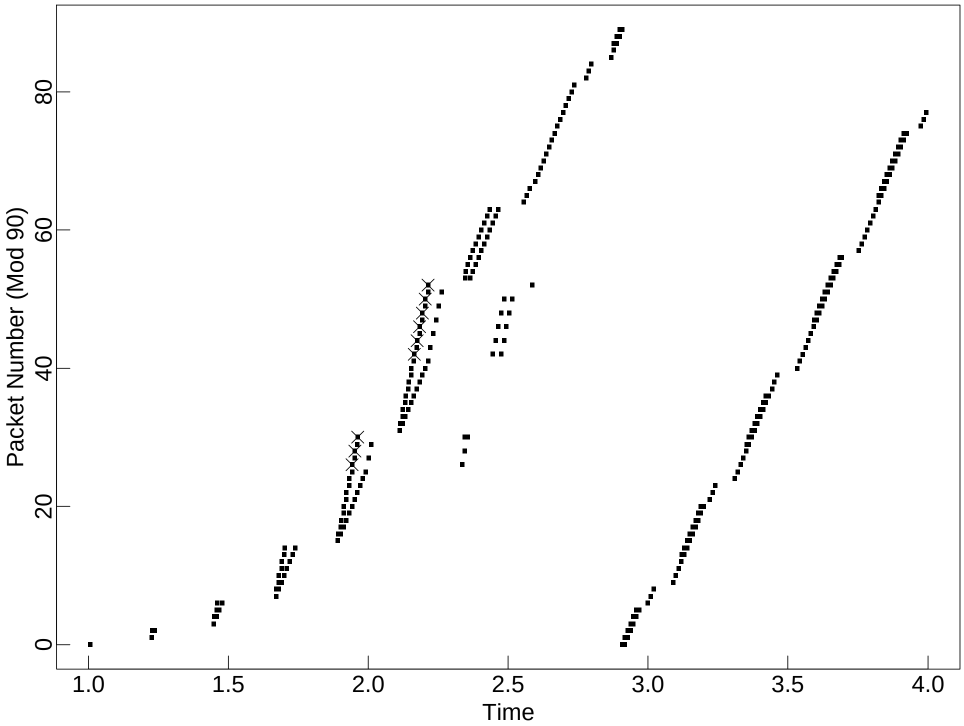

Figure 22: Fast Recovery with Selective Acknowledgements

```
renotcp, tcp, satcp [ window=28 ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
ftp conv from satcp [start-at=1.0] at s1 to sasink at k1
```

This test shows the Fast Recovery algorithm with Selective ACKS, with a moderate number of packet drops.

The Selective ACK packets in our simulator are not limited to a particular size. Our simulator has a parameter that determines the number of "holes" that can be reported in the Selective ACK packet; for this simulation that parameter is set to 10.

This test is run on tcpsim with "csh test14.com".

9 Random Drop gateways

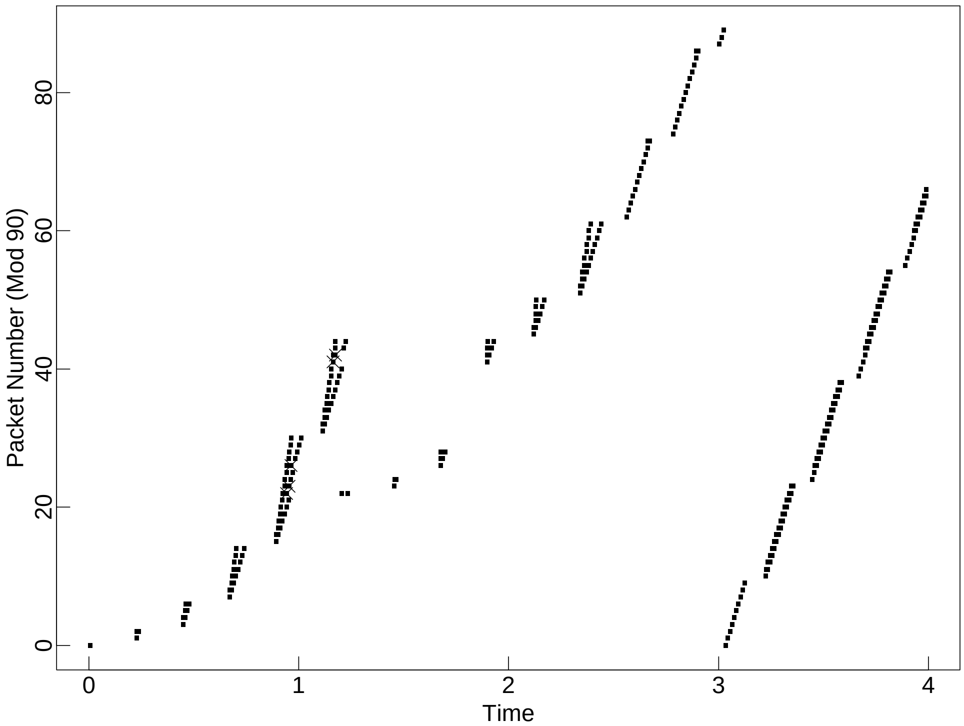

Figure 23: Random Drop gateways.

```
renotcp, tcp [ window=50 ]
bq [ dropmech=random-drop ]
edge s1 to r1 bandwidth 8Mb delay 5ms
edge r1 to k1 bandwidth 800Kb  delay 100ms
    forward [ queue-size=6 ]
ftp conv from tcp at s1 to sink at k1
```

This test shows a Random Drop gateway. With a Drop Tail gateway with the same scenario, the gateway drops every other packet of the slow-starting connection during times of congestion. This test differs from that in Figure 2 only in that in this test, the gateway uses a Random Drop rather than a Drop Tail packet dropping discipline. For a Random Drop gateway, when a packet arrives to a full queue, the gateway randomly chooses a packet to drop from the arriving packets and those already in the queue.

10 RED (Random Early Detection) Gateways

Simulations tests with Random Early Detection Gateways are shown in a separate document [F96].