# TCP and Successive Fast Retransmits

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

May 1995
(This is an expanded version of a note
released in October 1994.)

## 1 Successive fast retransmits in current TCP implementations

In this note we point out a long-standing problem for current Tahoe and Reno TCP implementations that results from invoking Fast Retransmit more than once in one roundtrip time. The problem is illustrated by packet trace from simulations. We have seen the same behavior in packet traces of TCP traffic on the Internet.

Given current TCP implementations, for a TCP connection with a large congestion window and multiple nonconsecutive packet drops within one window of data, it is possible for the TCP source to execute the Fast Retransmit procedure twice for one window of packets. For Tahoe TCP, this can occur when there are at least two nonconsecutive runs of packet drops in one window of data.

## 2 A simulation of a packet-based network

See Figure 2. First, the Tahoe-style TCP source receives three duplicate ACKs, infers a dropped packet, and begins slow-start. At 2.9 seconds, during the slow-start triggered by a Fast Retransmit, the congestion window is 4 packets, and the source retransmits packets 131 through 134, receiving four acknowledgements in return for packet 141. The first ACK for packet 141 causes the source to transmit packet 142. Immediately after that, three duplicate ACKs arrive acknowledging packet 141, triggered by the receipts of the retransmitted packets 132, 133, and 134. and the source uses the Fast Retransmit procedure to Slow-Start and to retransmit packet 142. The exact train of events after this is somewhat intricate, and we won't go through the details, but Figure 2 shows the pathological behavior that can result from multiple Fast Retransmits in one roundtrip time.

This problem is somewhat more difficult to duplicate in simulations with Reno implementations. With Reno implementations, the source essentially assumes that only one packet has been dropped, retransmits that dropped packet, and instead of waiting for the ACK to be received, continues transmitted new packets. For multiple packet drops in one roundtrip time, the Reno source often has to wait for a retransmit timer to recover (given the absence of Selective ACKs). And in some circumstances with Reno, the ability to have multiple Fast Retransmits in a single roundtrip time can avoid the wait for a retransmit timer timeout, in the absence of Selective ACKs. However, it is also possible for a second Fast Retransmit to be invoked from duplicate ACKS received from packets retransmitted during the slow-start triggered by the retransmit timer timeout. This leads to problems similar to those shown in Figure 2.

*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.

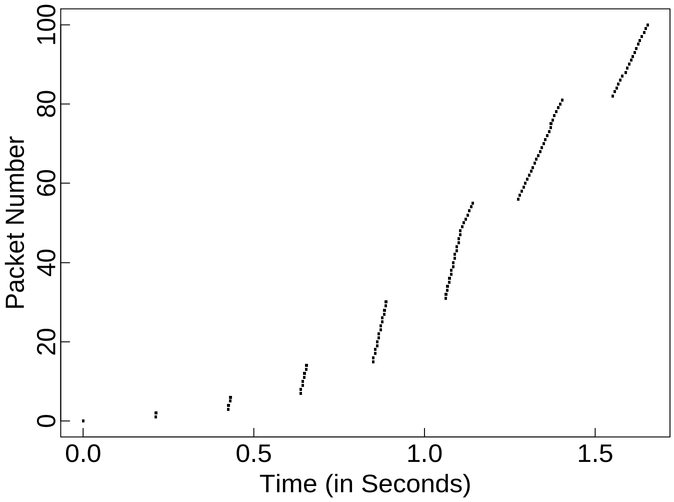


Figure 1: The first 100 packets

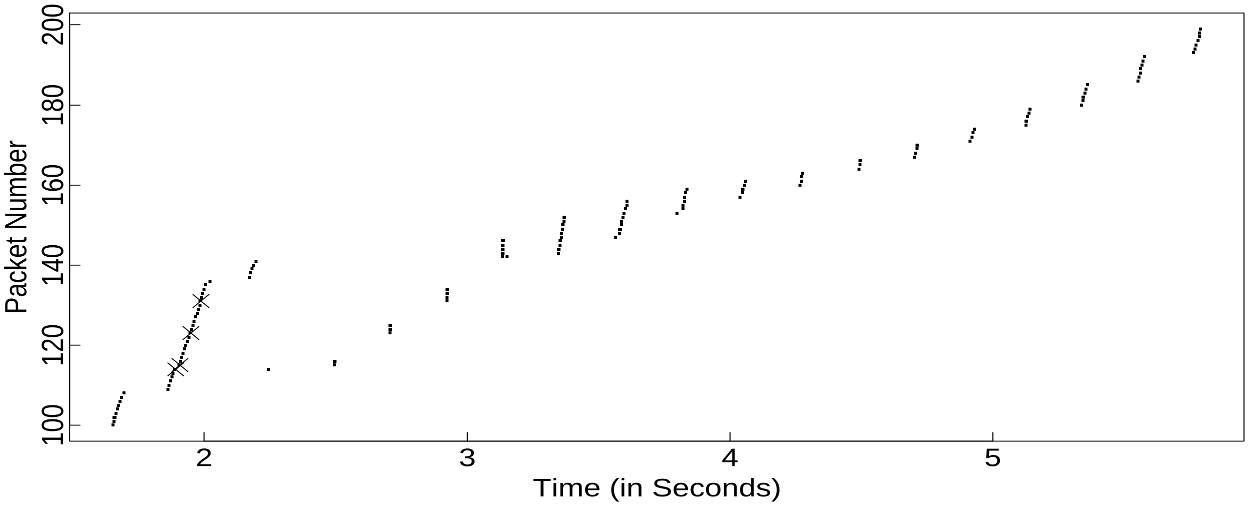


Figure 2: The second 100 packets, with Tahoe TCP

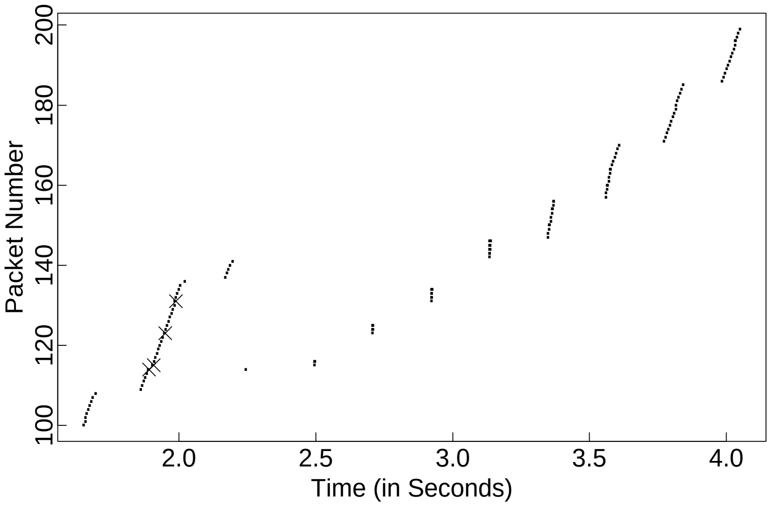


Figure 3: The second 100 packets, with Tahoe TCP modified not to allow multiple Fast Retransmits from one window of data.

# 3 A simulation of TCP over ATM

A second illustration of problems of multiple Fast Retransmits comes from Tim Dwight [D95], from simulations of TCP/IP over ATM.

Figure 4 shows the pathological behaviour that can result from multiple fast retransmits. The dots show packets and the open boxes show acknowledgements. The x-axis shows the time that packets were transmitted on an interior link in the simulated network. The dropped packets can be inferred from the trace.

The first Fast Retransmit in Figure 4 results from three dup acks for packet 25. The second Fast Retransmit results from three dup acks for packet 42, the last packet transmitted before the first Fast Retransmit was initiated.

Because the x-axis shows the time that packets appeared on a link within the network, the sequence of events at the sender has to be inferred from the graph. After the first Fast Retransmit, when the sender's congestion control window reaches four, the sender transmits packets 29-32. The sender receives an ACK for packet 29, and transmits packets 33 and 34. Next the sender receives an ACK for packet 30, and retransmits packets 35 and 36. Finally, the sender receives two dup acks for packets 42 (as responses to packets 31 and 32). At this point the congestion window is 6, and the sender transmits packets 43-48.

When the receiver receives packets 33-36, the receiver sends four dup ACKs for packet 42. These dup ACKs trigger the second Fast Retransmit and the sender reduces the congestion window to 1 and transmits packet 43. The receiver next receives packets 43-48, and returns ACKs. Immediately after the second transmission of packet 43, the sender receives the ACK from the first transmission of packet 43. The sender increases the congestion window to 2 and sends packets 44 and 45. The trace continues to unfold in this fashion.

In this case, the second Fast Retransmit triggered by dup acks for packet 42 ultimately leads to a succession of fast retransmits. There is a Fast Retransmit every roundtrip time, the congestion window never gets larger than 6 packets, and every packet is transmitted twice. In this case, this pathological scenario will continue indefinitely.

# 4 Recommendations

One fix to the problem of multiple Fast Retransmits is not to treat duplicate ACKs that acknowledge packets from the same window as packets from a previous Fast Retransmit as an indication of continued congestion.

In the Tahoe TCP implementation in our simulator, the fix was done using an extra variable *high_seq* to record the highest sequence number outstanding when the TCP initiated a Fast Retransmit or responded to an ECN (Explicit Congestion Notification [F94], such as a Source Quench message, or the Explicit Congestion Notification bit implemented in our simulator in packet headers) or a retransmit timer timeout. Duplicate ACKs that did not acknowledge data higher than this sequence number, not necessarily being an indication of congestion, would not trigger a Fast Retransmit. Once the TCP source transmitted a packet higher than the variable *high_seq*, then the variable would be disabled (e.g., set to zero) until the next congestion event.

In a Reno TCP implementation, the issues are slightly different. One possilibity would be to set the variable *high_seq* when the TCP source responds to an ECN or to a retransmit timer timeout, but not to set it when TCP initiates Fast Retranmit/Fast Recovery. This would still allow multiple Fast Retransmits during Fast Recovery, but would prevent the sequence of a Fast Retransmit/Fast Recovery, a timeout, and then a second Fast Retransmit/Fast Recovery for the same window of data.

The disadvantage of this fix is that, for both the Tahoe and the Reno cases, and for acks that do not acknowledge data greater than *high_seq*, the TCP source cannot distinguish duplicate acks resulting from retransmitted packets that had previously been correctly received by the receiver, and duplicate acks resulting from packet losses. In the absence of Selective ACKs, it is inevitable that any fix would rely on incomplete information, and therefore would occasionally result in suboptimal behavior.

Thus, the most robust and appropriate fix to this problem would be to implement Selective ACKs. The problem of multiple Fast Retransmits described in this section only occurs because the source retransmits packets that have already been correctly received by the receiver. With Selective ACKs, this behavior could generally be avoided.

# References


[D95] Dwight, Tim, private communication, 1995.

[F94] Floyd, S., TCP and Explicit Congestion Notification, ACM Computer Communication Review, V. 24 N. 5, October 1994, p. 10-23.

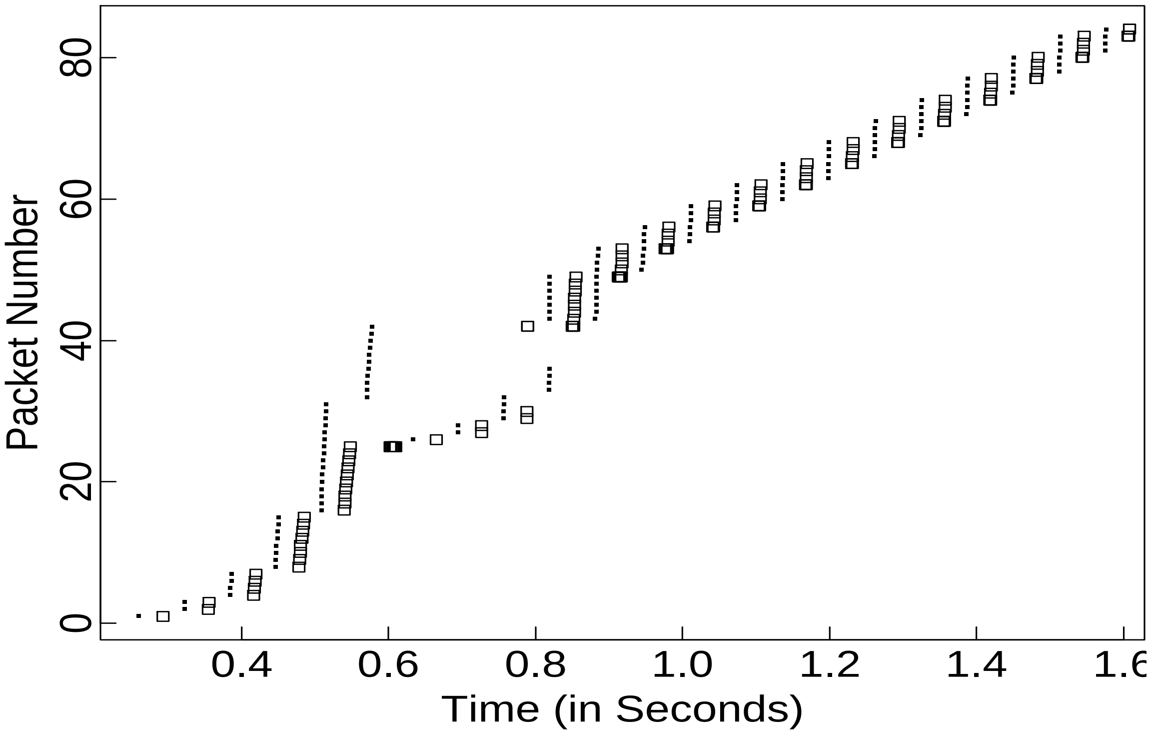


Figure 4: Multiple Fast Retransmits