CS 460: Congestion Control Part 1

Kevin Haroldsen

1 Introduction

Congestion control is a vital part to making TCP a practical protocol. Although participants could technically ignore congestion control and send what they desire, most do not. This report focuses on two popular congestion control mechanisms: Tahoe and Reno.

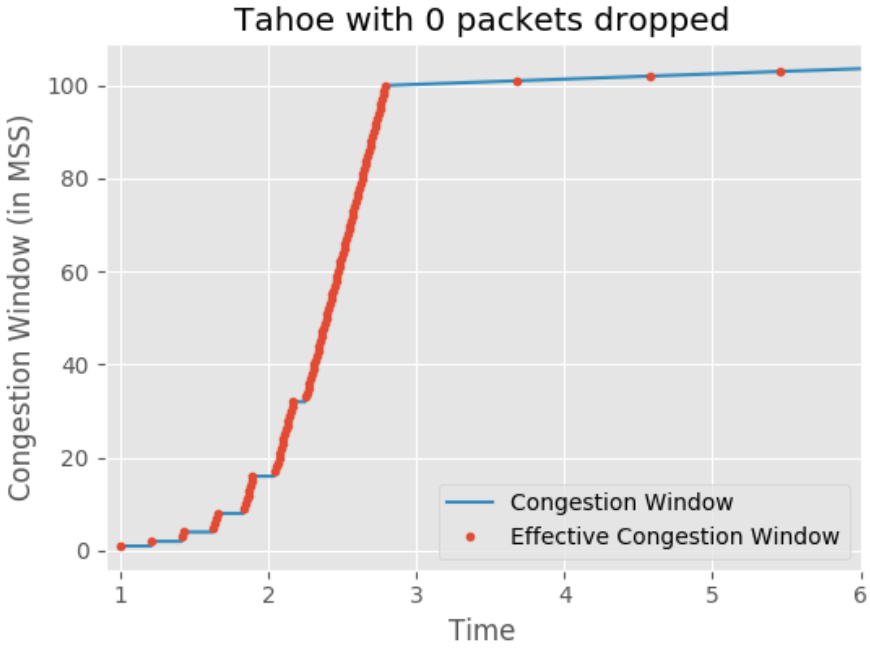
Their performance will be simulated with the Bene network simulator. Different packet loss scenarios will be compared. The way each loss scenario and congestion control mechanism scales with certain types of loss will be explored. These scenarios will include a graph of the congestion window over time, a plot of the packet sequence, and a discussion on the results of the simulation.

In this report, it's assumed that the Maximum Segment Size (commonly abbreviated as MSS) is 1,000 bytes. The Tahoe/Reno threshold is initially set to 100,000 bytes. The simulation is run with a two node network with a bidirectional link. This link has a bandwidth of 1 Mbps, and a latency of 100 ms. The receiving window size is also assumed to be arbitrarily large. The "fast retransmit" technology of detecting duplicate ACK packets is set to trigger at 3 duplicate ACKs (on the fourth ACK received with the same ACK number). The timeout for the TCP retransmission timer is set to 2 seconds. Other methods of loss detection, such as the ECN bit, are not explored.

2 TCP Tahoe

2.1 Slow Start

As expected, when no packets are dropped, the threshold of 100,000 bytes is reached very quickly.

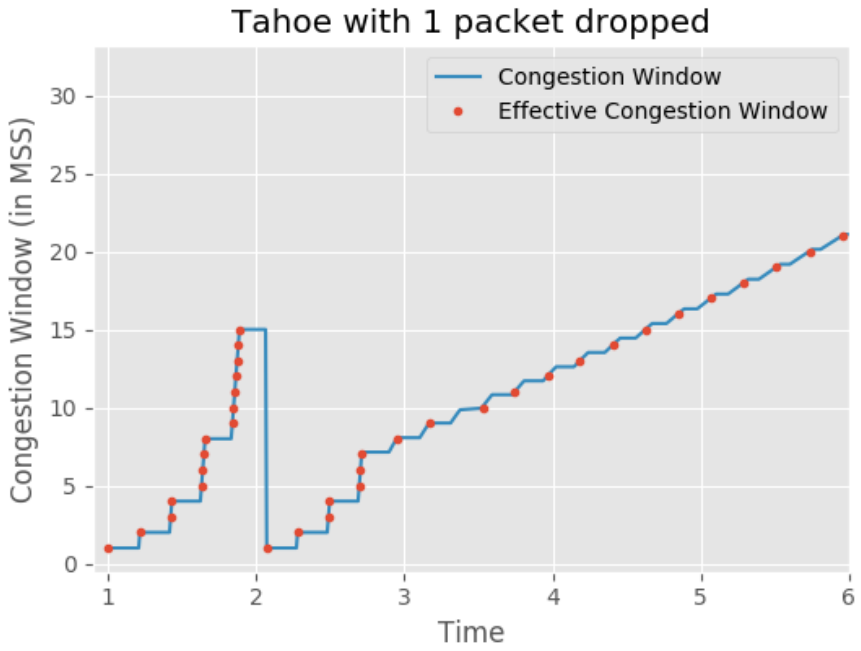


This graph, and the other congestion window graphs in this paper, includes two sets of data points. The Congestion Window is the internal counter of how many bytes that could theoretically be sent. The Effective Congestion Window is this value aligned to a Maximum Segment Size (MSS) boundary. A red data point is shown every time the effective congestion window changes, so congestion window changes outside of MSS boundaries can be seen. The values are also shown in units of MSS bytes, to allow simpler visualization.

In this case, the congestion window increases exponentially until it reaches the threshold value of 100,000 bytes. At that point, additive increase begins, and the congestion window gradually increases by one.

2.2 One-Packet Loss

In this case, the 14th segment (sequence number 14,000) is dropped. This results in the following congestion window over time:

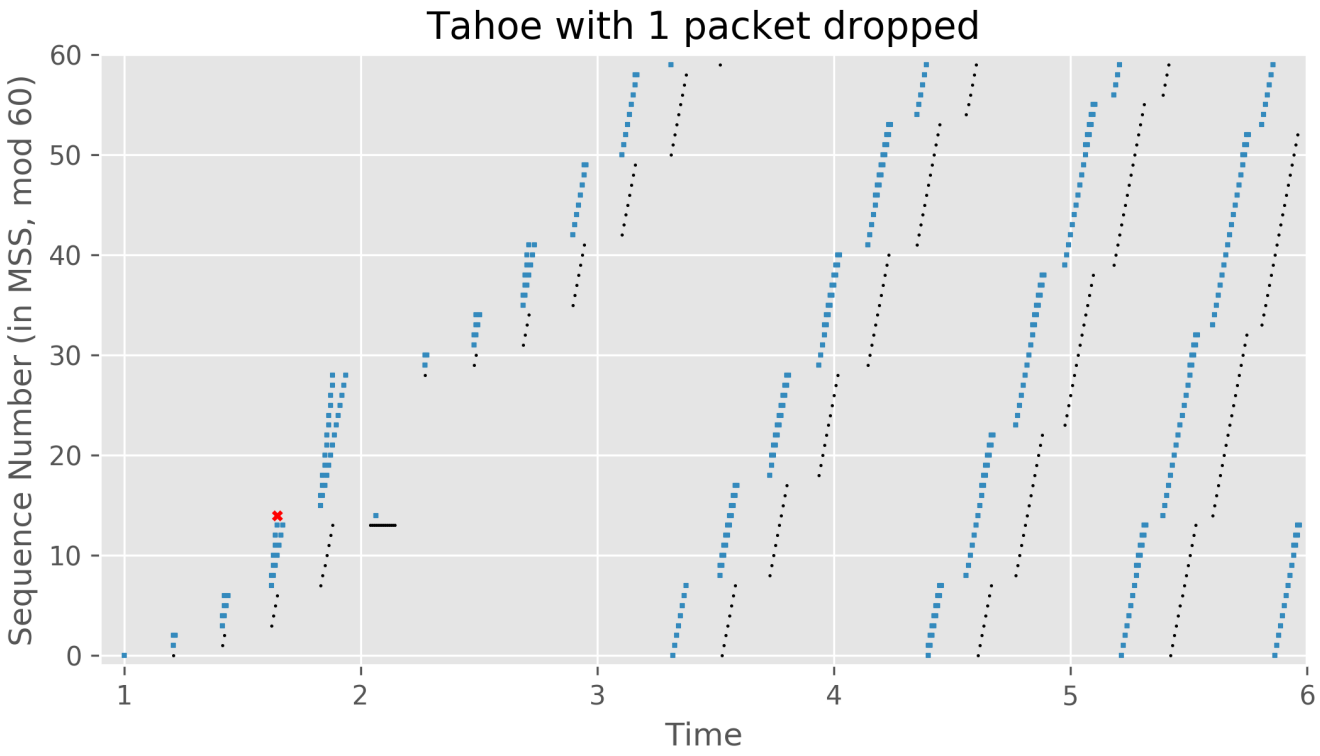


At segment 14, the size of the congestion window is 15,000. The loss event can be seen at the 1.064 seconds. Then, the congestion window is dropped down to 1. The new slow start threshold is set to 7,000. This is calculated by dividing the current congestion window by two, and aligning on the MSS boundary.

It can be seen that after the loss event, slow start begins again. This ends, and additive increase starts, when the congestion window hits the new threshold of 7,000 (7 MSS).

Also to note is the area from around 3.1 seconds where the effective congestion window appears to stay the same. This is due to a misalignment of the packets being received and the MSS boundary. The congestion window that includes partial units of MSS highlights that the value changes, but barely not up to the needed value to reach the boundary.

Altogether, this results in the following sequence graph:

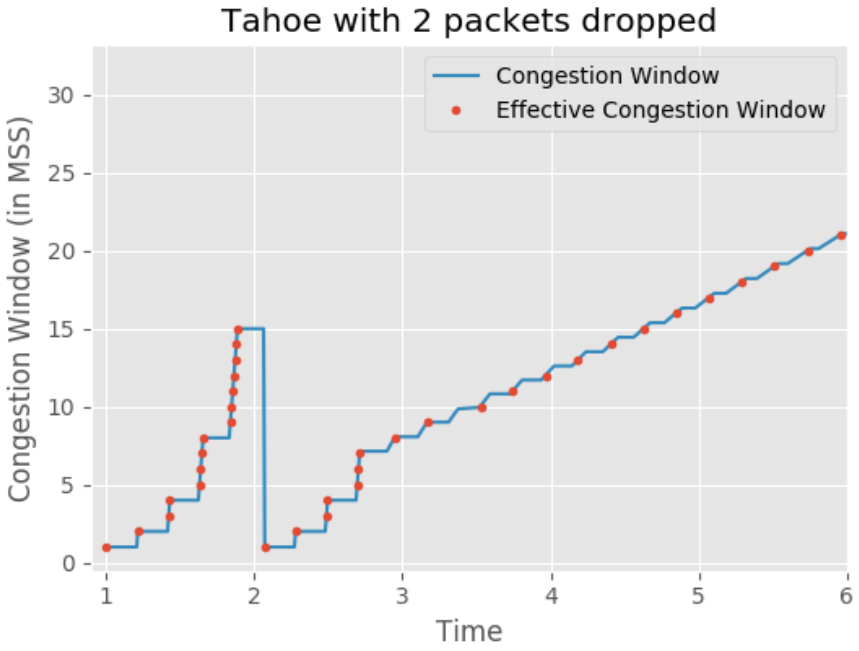


The above graph indicates that many duplicate ACKs are received for the 14th segment. This causes fast retransmit to be triggered, the congestion window and threshold to be resized, and that segment to be resent. Once the ACK for the 14th (and additionally sent packets) is received, it can begin sending again, this time in slow start.

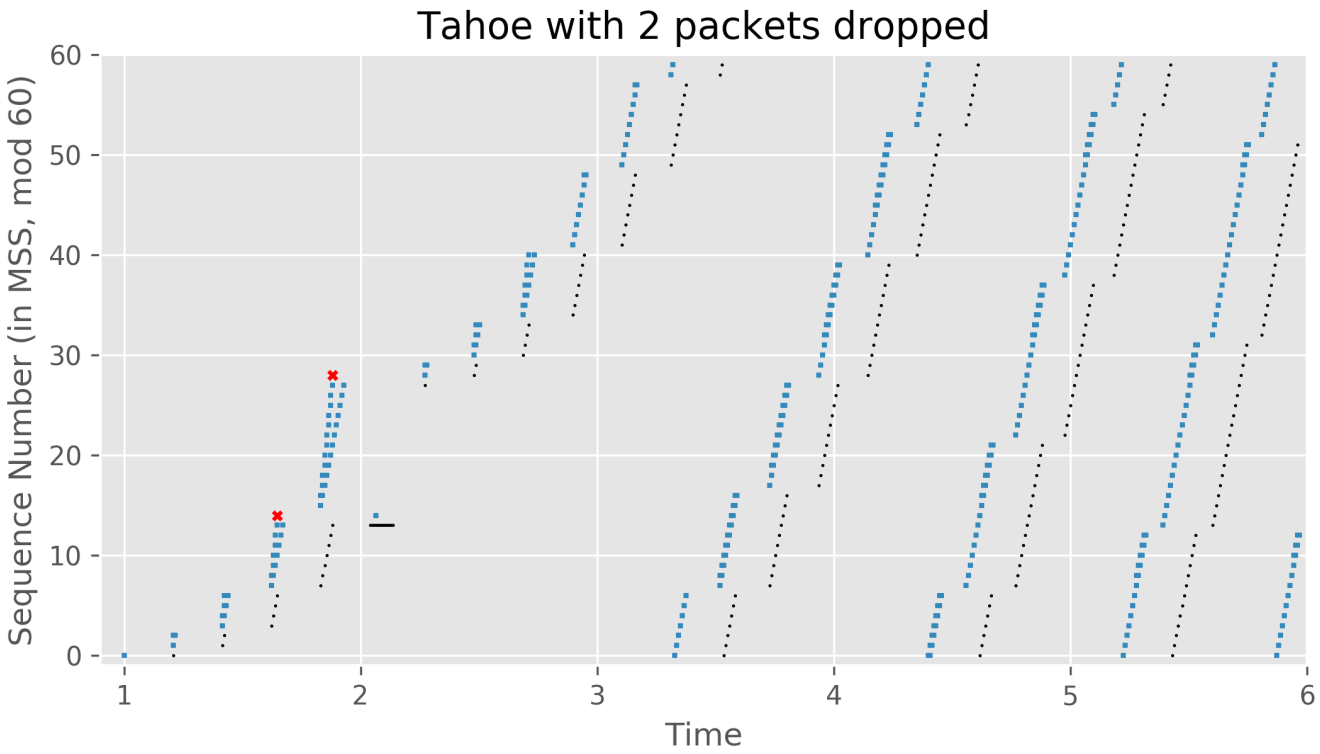
With the addition of friendlier coloring, the output is the same as seen in the SACK paper.

2.3 Two-Packet Loss

In this case, the 14th and 28th segments are dropped. This results in the following congestion window over time:



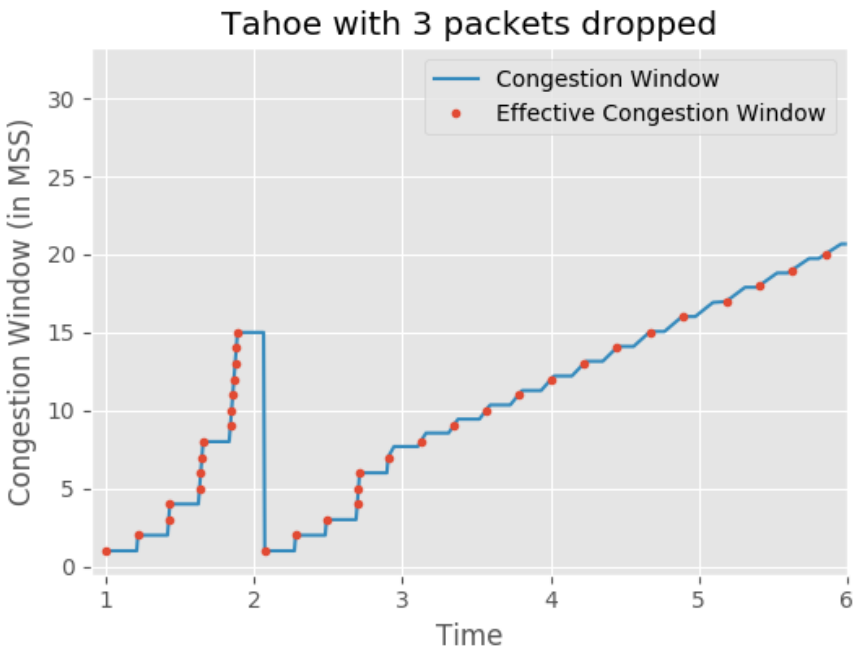
Notice that this looks extremely similar to the congestion window for one packet dropped. The reason for this may be a bit easier to understand if the sequence diagram is viewed:



In this case, the second packet is dropped before the first packet's loss is recorded, when fast retransmit runs. Thus, a loss is only detected once, and so the congestion window and threshold are only reset once, resulting in the same congestion window as for 1 packet dropped. This is primarily due to the packets chosen to be dropped.

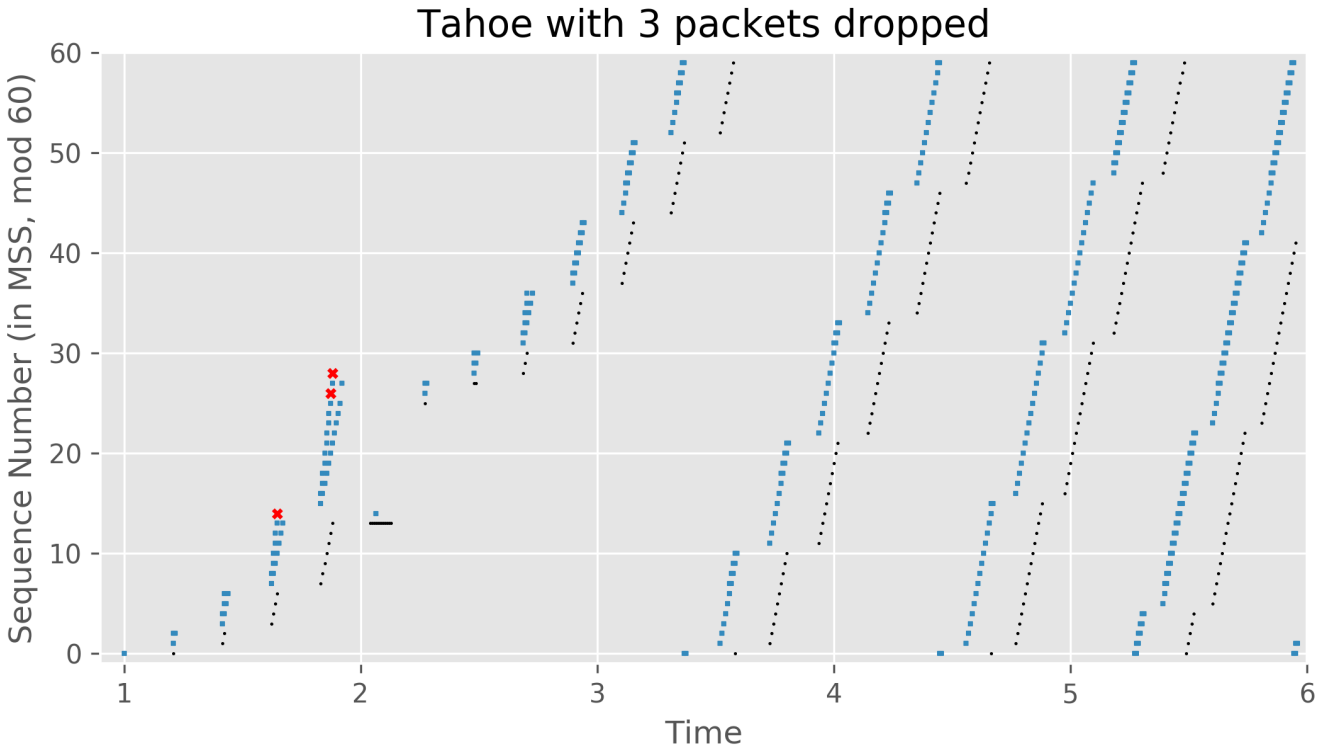
2.4 Three-Packet Loss

In this case, the 14^{th}, 26^{th}, and 28^{th} segments are dropped. Because an extra packet, the 26^{th}, is dropped, the slow start increase is less than in the two-packet loss. This causes the slow start to only increase the congestion window by one MSS before hitting the threshold. So, the partial alignment of the congestion window is slightly different, causing it to be off by one as compared to the two-packet loss. This results in the following congestion window over time:



The only significant difference between this graph and the others is what happens after the first loss event, but before the threshold is reached.

This can also be seen in the sequence diagram:



2.5 Tahoe Summary

Tahoe is not optimal. However, it is clearly able to recover well enough to handle even a carefully constructed loss sequence, and prevent timeouts, even if recovery is mediocre.

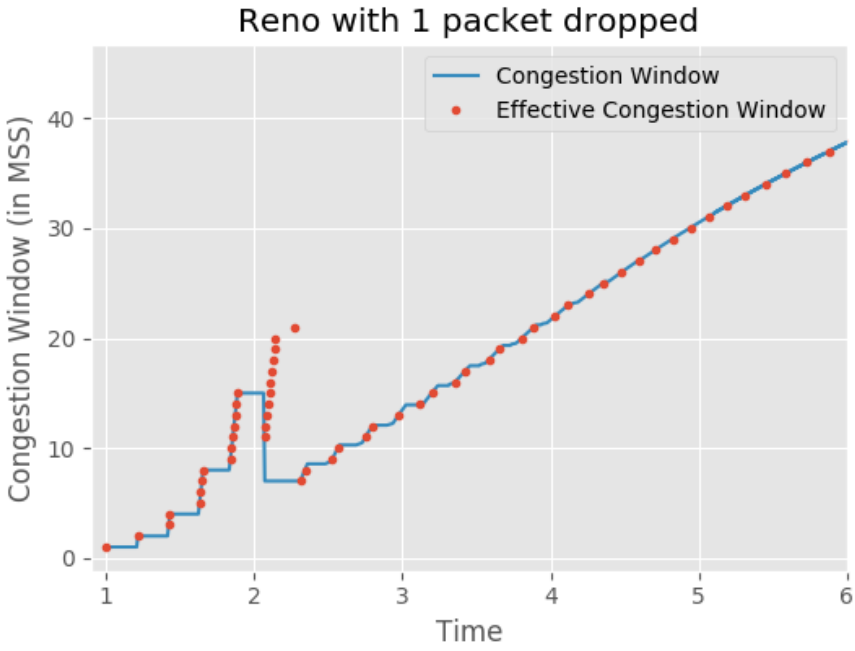
3 TCP Reno

3.1 Slow Start

Because no loss occurs, the result for this simulation was identical to Tahoe's.

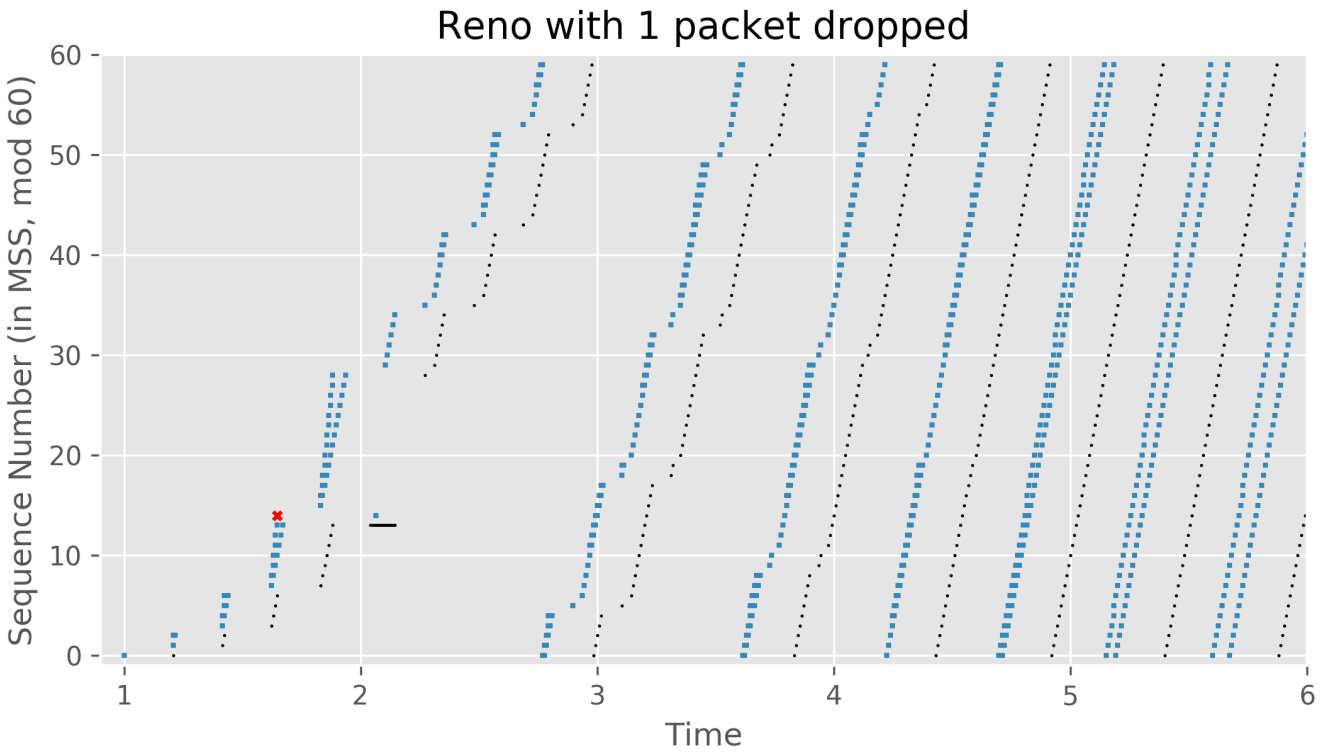
3.2 One-Packet Loss

In this case, the 14th segment (sequence number 14,000) is dropped. This results in the following congestion window over time:



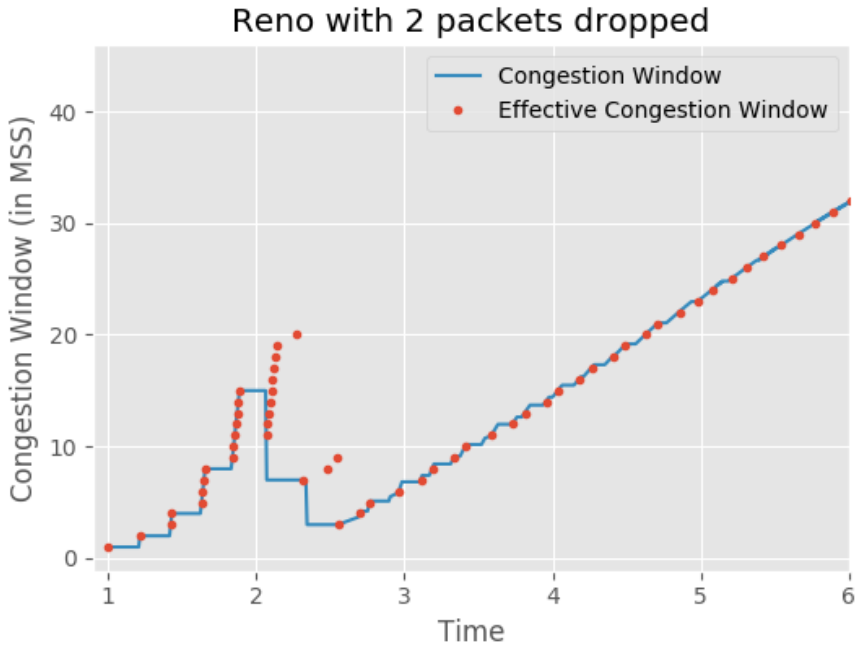
This plot may need some additional explaining. In the case of Reno, due to an implementation detail, the Effective Congestion Window is not necessarily the same as the Congestion Window aligned to an MSS boundary. What is occurring here is a phenomenon of fast retransmit. Each duplicate ACK "confirms" that another segment past the current dropped segment was correctly sent to the recipient. Effectively, the current window of data that can be sent is "extended" by the number of duplicate ACKs received. Thus, Reno treats the allowed congestion window during fast recovery to be more than half of the previous congestion window. The issues with this implementation can be seen when further loss occurs.

Of further interest is the packet sequence graph. It can be seen that even before the ACK for the first lost packet is received, that additional packets are being sent as additional duplicate ACKs come in. There's a slight misalignment of the first packet in each stream a few times; however, the link reaches full utilization relatively quickly, especially when compared to Tahoe. This closely matches the behavior shown in the SACK paper.

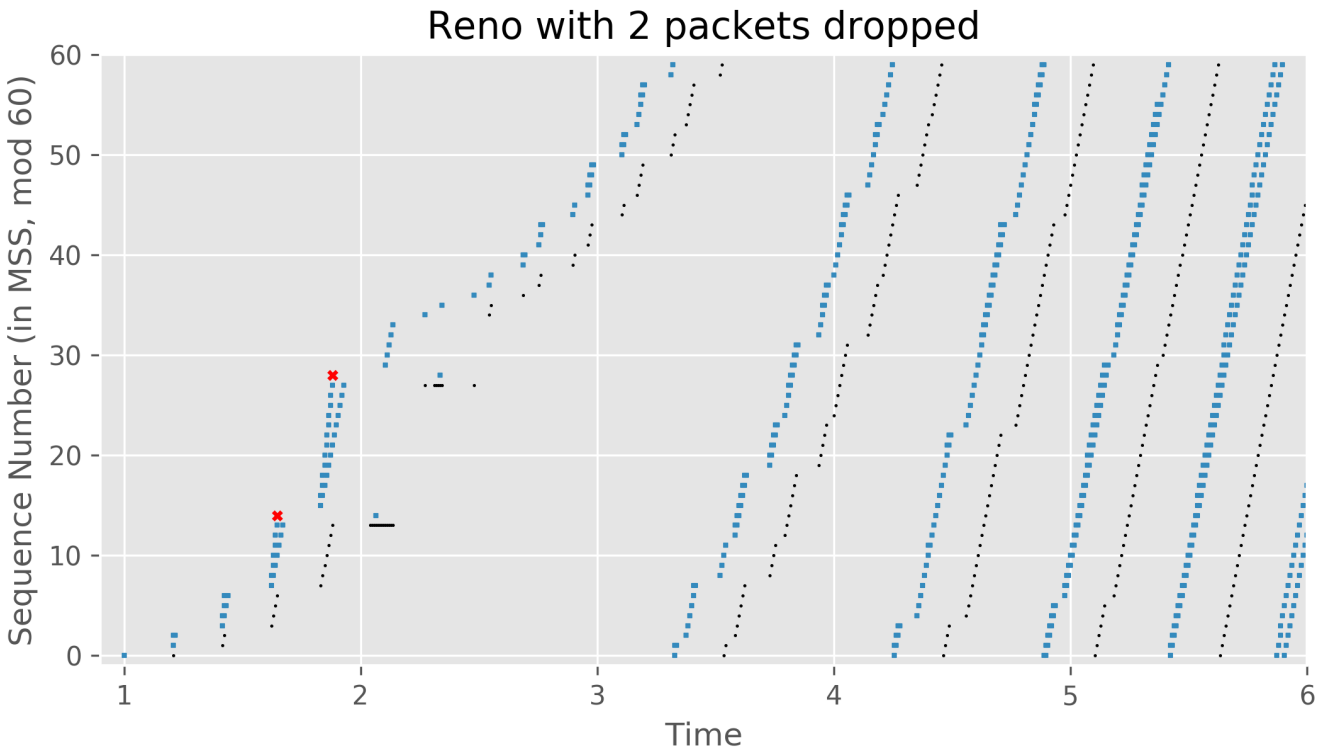


3.3 Two-Packet Loss

In this case, the 14^{th} and 28^{th} segments are dropped. This results in the following congestion window over time:



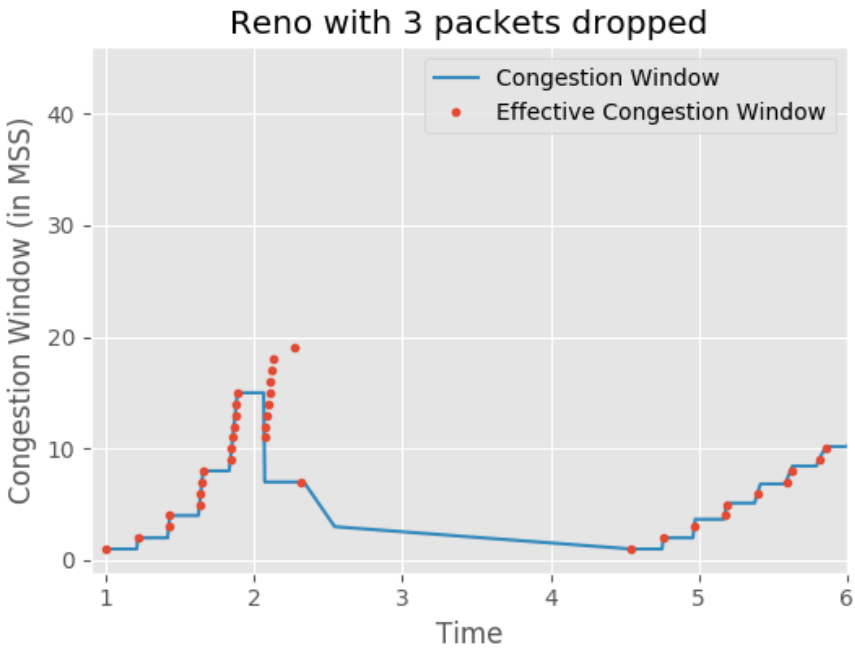
A similar phenomenon as in the one-packet loss can be seen in this graph, with the effective congestion window being separate due to fast recovery.



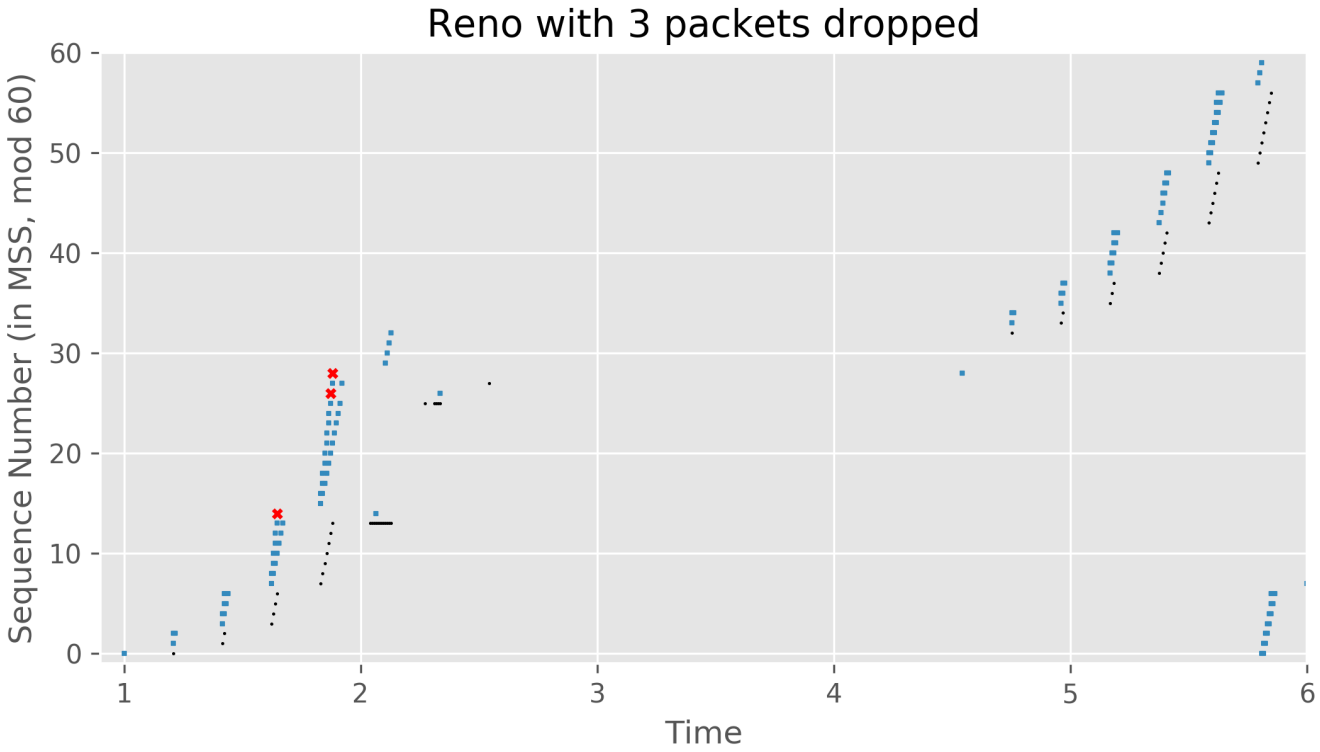
When compared to Tahoe, Reno does not perform as well in the short-term in this case, and "stumbles" to recover. Reno eventually approaches full utilization beating Tahoe, but it takes a little longer to recover than Tahoe, due to fast recovery (rather ironically). By now, it's clear that Reno altogether performs better with low packet loss, especially if these losses are close together.

3.4 Three-Packet Loss

In this case, the 14^{th}, 26^{th}, and 28^{th} segments are dropped. This results in the following congestion window over time:



Reno is not able to recover properly from the constructed packet, causing fast retransmit to not occur and the retransmission timer to fire. This is, by far, the most interesting result, even if expected.



The reason for the timeout is due to the sneaky structure of the packet loss.

First, the loss of the 14th segment is constructed so seven ACKs are received. These seven ACKs grow the congestion window to 14. As each of these ACKs are received, additional segments can be immediately sent, so they are. This includes the dropped 26th and 28th segments. Eventually, we receive 3 duplicate ACKs through segment 13. This causes Reno to immediately resend segment 14 and enter fast recovery. The additional duplicate ACKs received while waiting for confirmation on segment 14 cause Reno to think it's OK to send the segments 29-32.

Then, an ACK through segment 25 is received as a result of the retransmit sent for segment 14. Because Reno has received acknowledgment of segment 14, fast recovery is exited, resetting the congestion window to 6. Next come another four duplicate ACKs through segment 25 from the four segments 29-32. Fast recovery is entered once more, and segment 26 is resent. The congestion in window is 6 segments, and segments 26-32 (6 segments) are currently outstanding. So, no additional packets can be sent. An ACK through segment 27 is received. However, there are still outstanding packets, but no more can be sent.

No more duplicate ACKs will be received, so now Reno must fall back on the retransmission timer. When that fires, not only is a huge amount of time wasted, but now the congestion window is set to 1 segment.

This structure of packet loss could certainly occur in highly lossy environments.

4 Conclusion

Overall, Tahoe is simple to implement, but struggles with even small data loss. Reno is much better at handling small errors and recovering quickly, but completely fails with certain loss constructions.

However, duplicate ACKs and transmission timers are not the only methods of detecting network congestion. The introduction of the ECN bit also provides a method for the network itself to report congestion. Although this is outside the scope of this report, different congestion methods treat the varying ways of detecting loss in distinct ways.

A small correction to Reno, called New Reno, fixes some of the major flaws shown with high-error links. All it needs to do is always send a new unsent packet from the end of the congestion window, to keep the transmit window full. It also assumes that a new ACK that doesn't fully ACK sent data points to a new hole, and sends the packet at that hole.

Another one of the modern solutions discussed in other papers is SACK. This allows the receiver to acknowledge discontinuous data that were correctly received. This allows the sender to know where the holes are, and to fill them.

In the end, although senders can ignore congestion control, it's better for everyone to use a cooperative congestion control algorithm that allows equal distribution of network resources to connections. We've moved on from Tahoe and Reno, but learning the fundamentals allows us to better understand modern congestion control methods.