Lab 3 Report

Congestion Control (pt. 1)

Kyle Pontius

1 Why Congestion Control

In our previous lab we implemented the TCP reliability feature. This enabled data to be sent from a source with the guarantee of having the data correctly and completely reassembled at the receiver, even with intermittent loss on the link. This implementation is extremely effective on a connection where there is only one sender and one receiver. Data can be freely sent back and forth without interference, or congestion, from other machines. However, when scaling TCP it quickly becomes obvious that network congestion is a serious concern that must be resolved if the network is going to continue to grow. One of the most serious symptoms of network congestions is data loss, which increases exponentially as queue size on routers in the network approach 100%. As routers approach this level of congestion, almost all data becomes deadlocked and the network can no longer function. To address this problem, we use TCP's congestion control.

2 How Congestion Control Works

Congestion control functions on the assumption that all machines on the network are running this same protocol. If this is the case, the machines will approach an optimal state where bandwidth is shared fairly. Obviously there are certain conditions (such as round-trip time, packet size, and link speed) that make a difference in the real world. Congestion control relies on a few different features to be effective in controlling traffic on the network: (1) Dynamic congestion window, (2) Additive Increase with Multiplicative Decrease, (3) Slow Start, and (4) Fast Retransmit.

2.1 Dynamic Congestion Window

The congestion window used in TCP is central to controlling the rate at which packets are sent. The window size itself represents the number of packets that can be outstanding (packets sent, but not yet acknowledged). We implemented the congestion window in Lab 2, and we're modifying it in Lab 3 to be dynamic. Most, if not all of TCP's congestion control tools are rooted in adjusting the congestion window to maximize throughput and minimize congestion on the network. On connections where bandwidth is readily available, a large window is desirable as it allows the user to send more data at a time. However, on connections where bandwidth is at a premium, smaller windows help to minimize congestion in the network by preventing a large number of packets from being outstanding, for each machine.

2.2 AIMD

In TCP Tahoe, after any loss event (either due to timeout or 3 duplicate ACKs) the congestion window size is dropped to one maximum packet size (MSS). This the multiplicative decrease portion of AIMD. The threshold, which represents the point at which Slow Start stops and Additive Increase should begin, is dropped to half of the window size when the loss occurred. Once window size reaches that threshold again, additive increase takes over and slowly increases the size of the window until loss occurs again. In our implementation of additive increase, we increment the size of the window by ***MSS * new_bytes_received /***

window_size. On lossy connections, this allows the user to get very close to the point where data is lost before the loss actually occurs, maximizing the time spent sending data at the highest rate possible.

2.3 Slow Start

The goal of slow start is to quickly ramp up transmission speed, enabling TCP to quickly recover window size following a loss event. Due to the dynamic nature of connections, congestion, and packet loss it's necessary to have this flexible and fast-recovering tool in order to get back up to speed quickly. Slow Start is especially helpful on systems with high bandwidth as slow start can promptly return them to maximum available speed. In our implementation, it works by incrementing the window size by the number of new bytes received in an ACK. For example, if my program sends the packet with sequence number 1000 and an ACK comes back for sequence number 4000, the congestion window is incremented by 3000. When graphing your solution, you should expect to see an exponential increase in the number of packets being sent. Again, Slow start is ended, and additive increase begins, when the size of the window is greater than or equal to the threshold.

2.4 Fast Retransmit

Fast Retransmit (FR) helps to counteract lost packets by proactively resending packets that are presumed lost. If the FR system wasn't being used, the alternative is to wait for the retransmission timer to fire, then resend the packet. This is costly due to the relatively lengthy wait required by the timer. FR works simply by reviewing the last four ACKs received and comparing the ***ack_number*** on each one with the current. If they all match, meaning the last three ACKs were the same sequence number as the current, then that packet is immediately resent and the timer is restarted. All duplicate ACKs after this point are ignored until the next ACK is received, acknowledging new data. At this point, TCP returns to Slow Start and functions normally again.

3 Loss Events

Managing loss events in TCP is a critical component in the protocol. Loss events are simply defined as either the retransmission timer expiring or receiving three consecutive duplicate ACKs. When a loss event is detected, the threshold is set to the maximum other either half the congestion window or the MSS. The congestion window is then reset to MSS.8

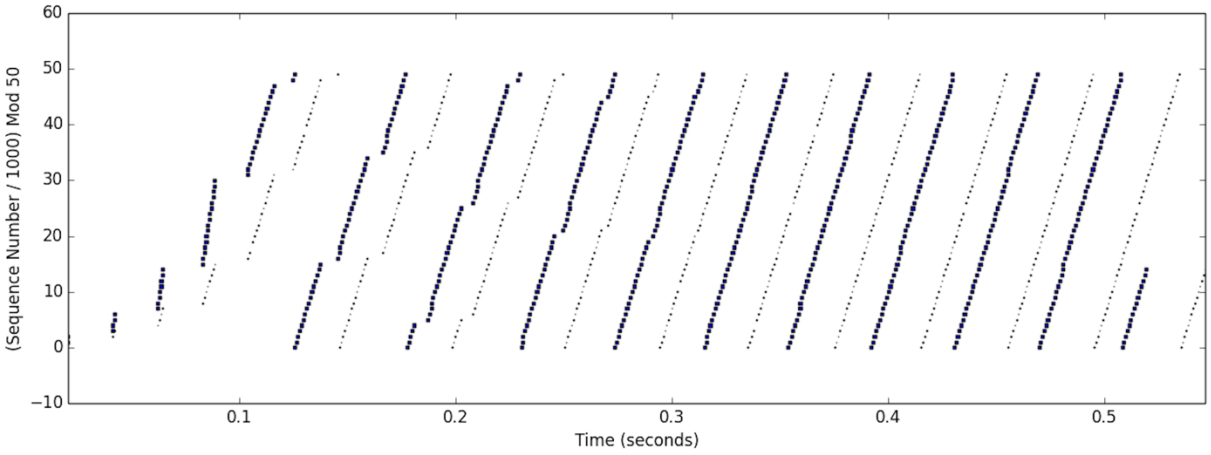
(see next page)

4 Results

My algorithm's behavior matched TCP Tahoe's behavior very well. In the following section I'll describe my network configuration and the results of my test. All tests were performed using the *internet-architecture.pdf*.

Loss: No packets
Link Speed: 10 Mbps (both links)
Propagation Delay: 10 ms (both links)

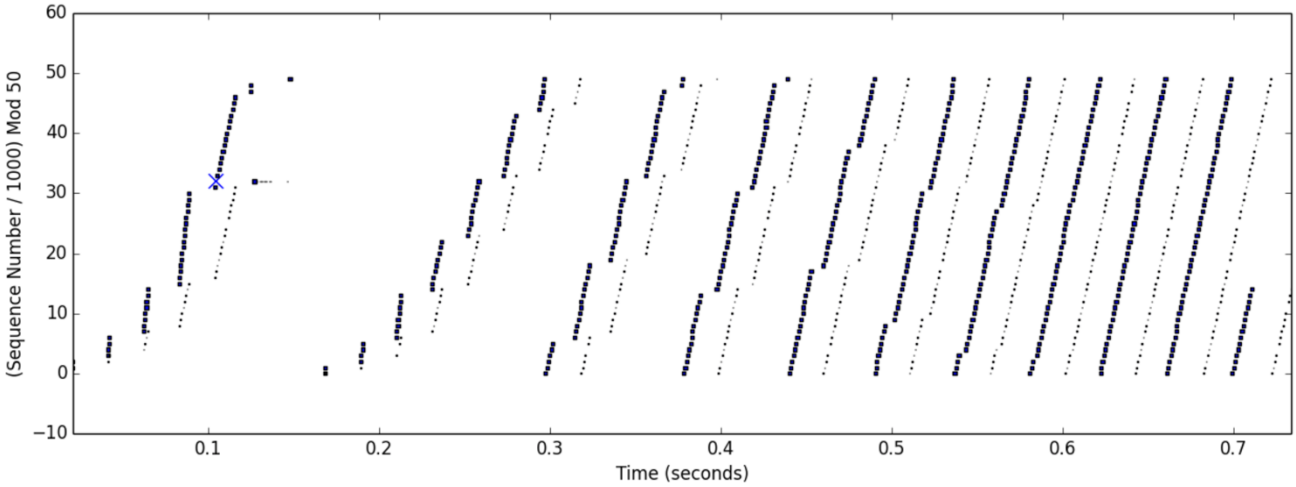
This configuration did not have any loss. From time 0.00 to about 0.13 the graph reflects Slow Start being employed to quickly work up to speed. At about time 0.13, additive increase takes over and continues to steadily increase the window size over time until the file has been completely transmitted.



(see next page)

Loss: 1 packet
Link Speed: 10 Mbps (both links)
Propagation Delay: 10 ms (both links)

In this configuration a single packet is dropped at sequence number 32,000. After the packet drops, all the ACKs are returned with ***ack_numbers*** for that packet. After 3 duplicate ACKs the Fast Retransmit is triggered and the packet is immediately resent. Please note, the ACKs return very quickly when the packet was resent, making the ACKs themselves difficult to see on the graph. After the packet is successfully received and ACKed, Slow Start begins again and exponentially increases the rate at which the packets are being sent. It's notable that the Fast Retransmit and Slow Start both help TCP to immediately recover from the loss and continue on without significant delay.



(see next page)

Loss: 3 packets
Link Speed: 10 Mbps (both links)
Propagation Delay: 10 ms (both links)

This scenario for TCP Tahoe drops three packets, then recovers gracefully to continue data transmission. Packets with sequence numbers 32,000, 40,000, and 41,000 were all dropped. My implementation successfully followed the pattern for TCP Tahoe, recovering almost immediately then using Slow Start again to get back up to speed. The execution that occurred was that the first packet was dropped, just like in the last test, where several ACKs were received that packet. After the 3rd duplicate ACK had been received Fast Retransmission resent the missing packet. After the missing packet was ACKed, the ACK for sequence number 40,000 came down and that was sent. Then the ACK for sequence number 41,000 came down. Finally, Slow Start was able to start again and the algorithm quickly got back up to speed. Slow Start ended and Additive Increase began around time 0.25.