Lab 4 Report

Nate Fox

1 Congestion Control

1.1 Class Variables

To implement congestion control, I started by changing some of the TCP class variables. I renamed the `window` variable to `cwnd` in order to match the variable names from the textbook, and I initialized `cwnd` to be one MSS. I also introduced the `ssthresh` variable to keep track of the slow start threshold. Finally I introduced the `ack_count` variable to keep track of duplicate ACKs.

1.2 Receiving New ACKs

When I receive ACKs for newly acknowledged data, I update my congestion window according to what state I am in. If my current `cwnd` is less than `ssthresh`, I consider myself in slow start mode and I increment `cwnd` by the number of new bytes acknowledged. If my current `cwnd` is greater than or equal to `ssthresh`, I consider myself in AIMD mode and I increment `cwnd` by (MSS * new_bytes / `cwnd`). I also reset my `ack_count` to 0.

1.3 Receiving Duplicate ACKs

When I receive a duplicate ACK, I increment my `ack_count`. If `ack_count` is three after I increment it, I retransmit without restarting the retransmission timer or updating the RTO.

1.4 Retransmitting

Whenever I retransmit, either due to the retransmission timer firing or due to three duplicate ACKs, I update the `ssthresh` and the `cwnd`. I set `ssthresh` to one-half of `cwnd` (to a minimum of one MSS) and I reset `cwnd` to one MSS.

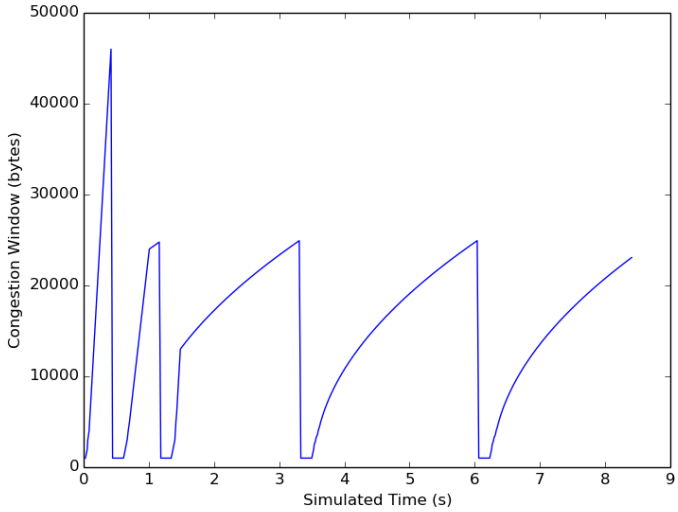
2 Basic Experiments

2.1 Setup

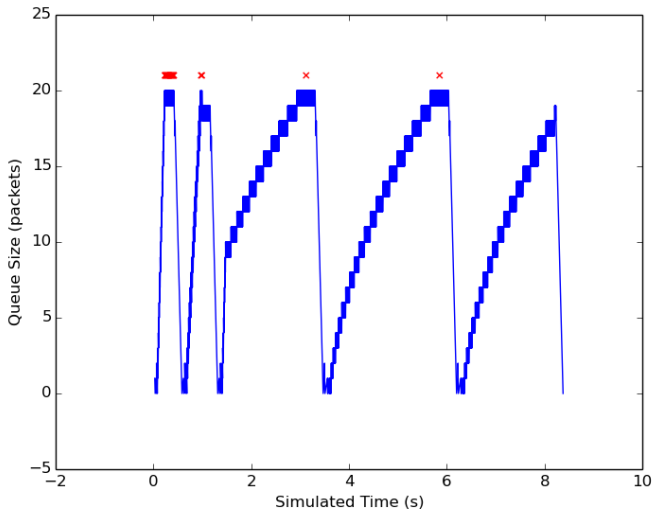
The network consists of two nodes with a single bidirectional link between them. The link has a 1 Mbps bandwidth and a propagation delay of 10ms. The queue size is limited to 20 packets (20,000 bytes). The initial `ssthresh` is set to 100,000 bytes.

2.2 One Flow

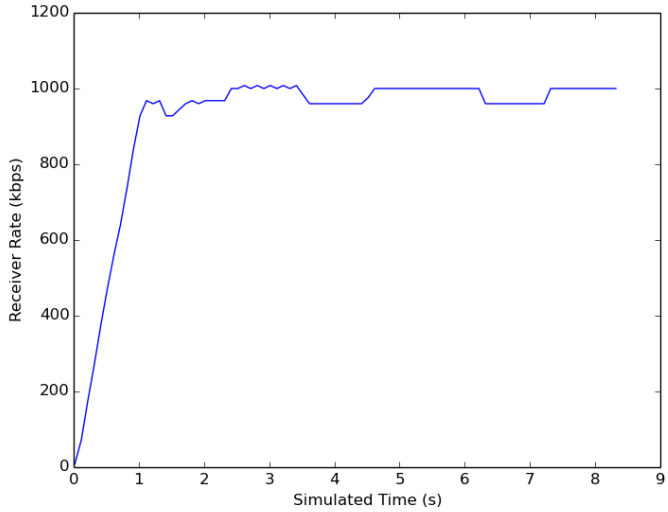
With the aforementioned setup, a 1 MB file is transmitted over the link.



The above graph shows the typical sawtooth pattern as the congestion window reacts to the congestion of the network. When a packet is dropped, the congestion window updates `ssthresh` and resets to 1,000 bytes. The congestion window grows exponentially in slow start mode until `ssthresh` is reached, at which point additive increase begins.



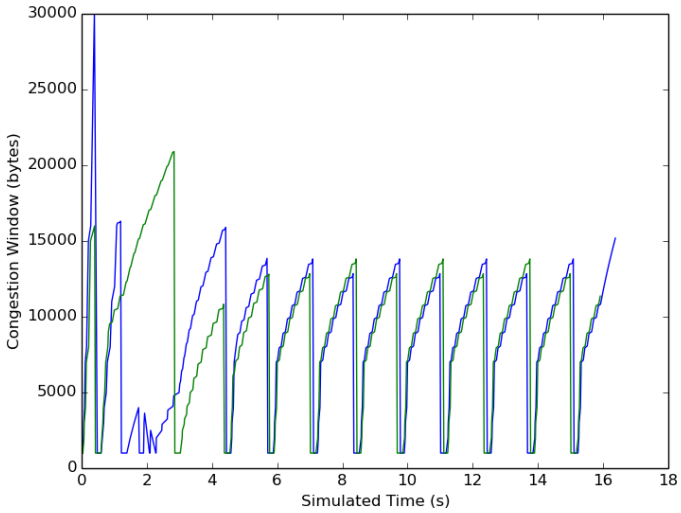
The above graph shows how the queue size changes over time. When packets are dropped due to queue overflow (marked by red X), the sender resets the congestion window to 1,000 bytes, which lessens the stress on the queue. For this reason, the queue size graph is comparable to the congestion window graph.



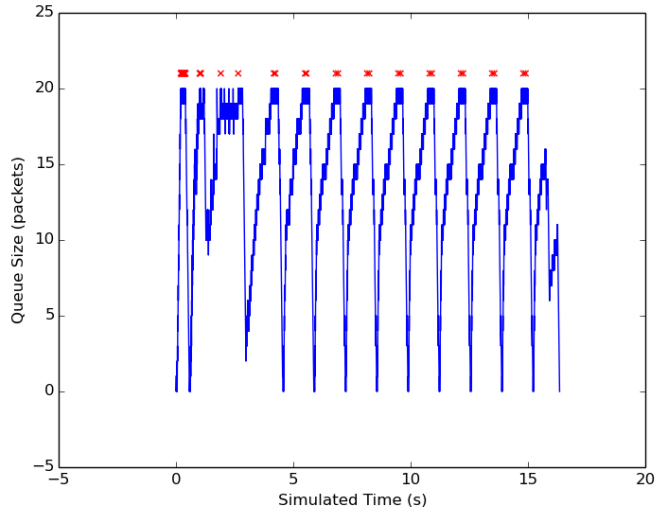
As shown in the above graph, the receive rate rapidly increases and then levels out at the full link bandwidth.

2.3 Two Flows

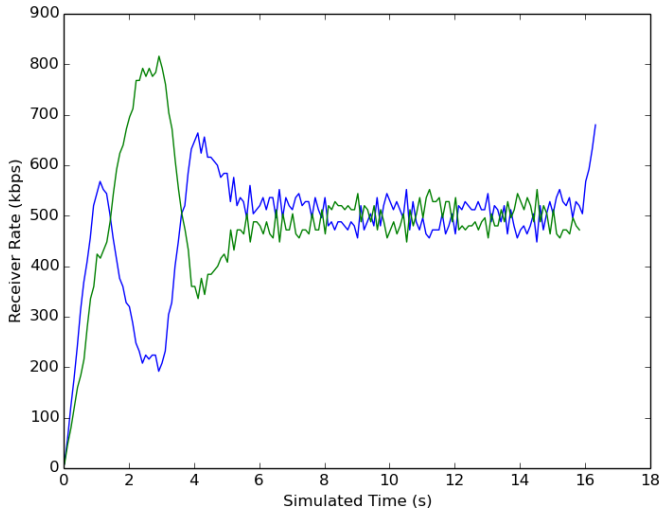
I wrote a new python script `two_flow.py` and a new shell script `run_two_flow.sh` to send two flows over a single link in parallel. I modified log messages from `tcp.py` to include information on which flow the log messages pertained to. I also modified `plot.py` to graph separate lines for each flow on the receive rate and congestion window graphs. A 1 MB file is transmitted over each flow on the link.



The congestion window graph above shows that both flows experienced the typical sawtooth pattern. They start out differently, but after a few seconds they adjust and follow the same pattern.



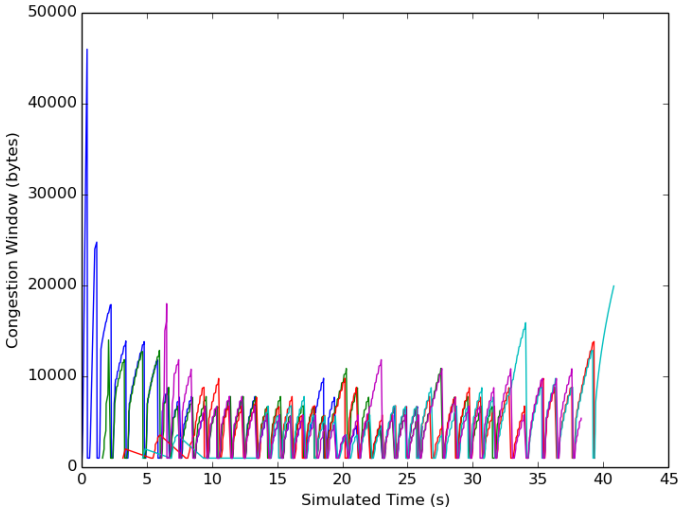
With two flows over the same link, the queue overflows a lot more. The queue graph still looks very similar to the congestion window graph, particularly after a few seconds when both flows' congestion windows follow the same pattern.



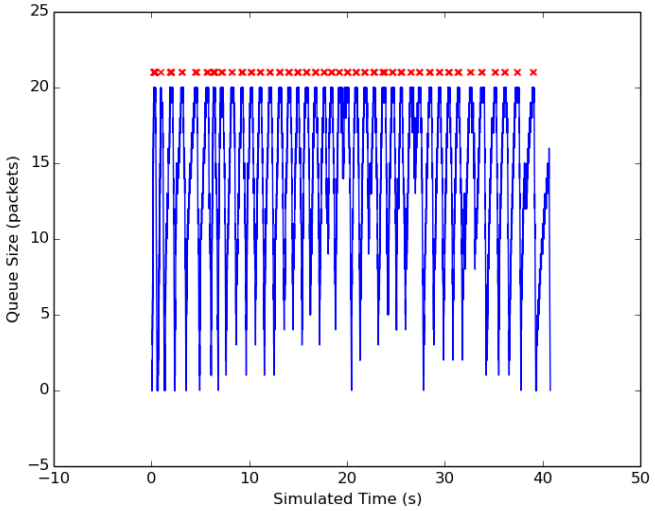
The first flow (blue line), detected packet loss before the second flow (green line), so the first flow drops to about 200 kbps and the second flow jumps up to use the remaining 800 kbps. Over time, the two flows converge around 500 kbps, which is half the link bandwidth. The file transfer finishes slightly earlier on the second flow, allowing the first flow to use more bandwidth as it finishes the transfer.

2.4 Five Flows

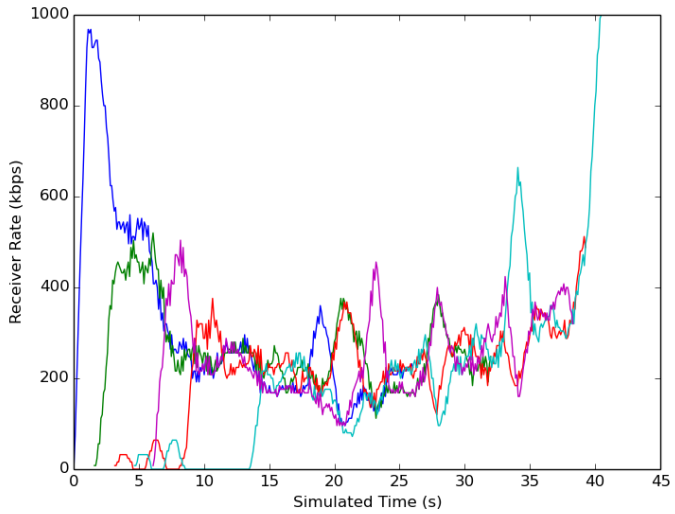
I wrote a new python script `five_flow.py` and a new shell script `run_five_flow.sh` to send five flows over a single link in parallel. A 1 MB file is transmitted over each flow on the link, with a 1.5 second offset between each flow.



The above graph shows a very condensed sawtooth pattern. With five separate flows each transferring a 1 MB file over a link with a queue that only holds 20 packets, packet loss occurs frequently, resulting in a low `ssthresh` and frequent `cwnd` resets.



As previously discussed, packet loss occurs frequently with such a high load.



The 1.5 second offset between flows lets the first flow (blue line) use the full bandwidth for a short time. When the second flow (green line) comes in, the two flows adjust to about 500 kbps each. Once all five flows come in, they all eventually adjust to 100-300 kbps each until the first flow finishes the file transfer. The remaining flows use the extra bandwidth, and at the end there is a single flow that gets all the bandwidth to finish its file transfer.

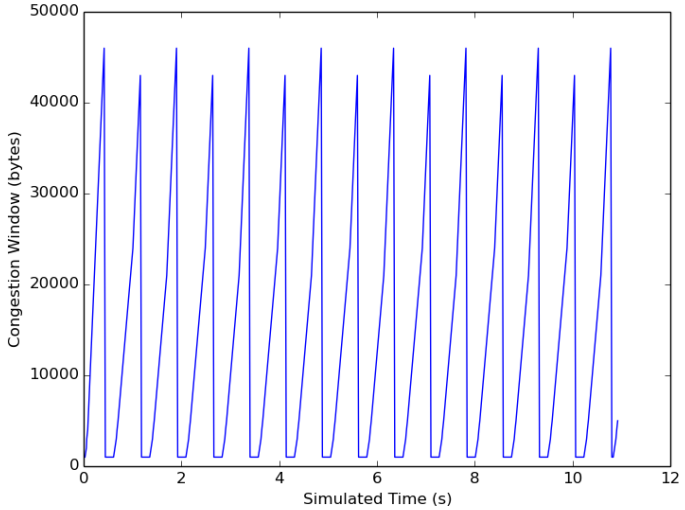
3 Advanced Experiments

3.1 AIAD

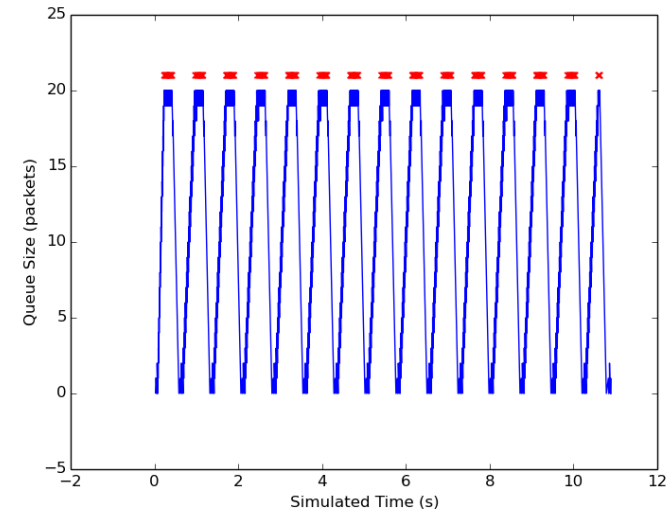
I changed my `tcp.py` code to use AIAD rather than AIMD:

```
def do_retransmit(self):
        # Set ssthresh and cwnd
        #self.ssthresh = max(self.cwnd/2, self.mss)
        self.ssthresh = max(self.ssthresh-self.mss, self.mss)
        self.cwnd = self.mss
```

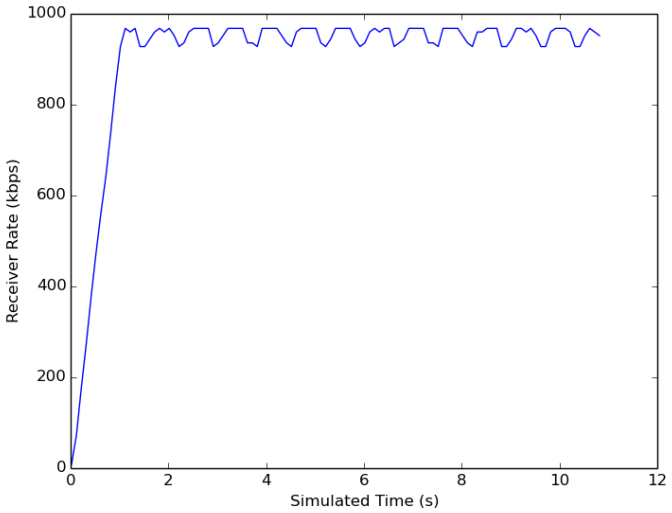
With the same network setup described in the previous section, I sent a 1 MB file over a single flow.



With AIAD, the congestion window does not follow the familiar sawtooth pattern. Instead, it quickly oscillates back and forth with no evidence that it will ever converge.



Since the congestion window osciallates back and forth so quickly, the queue is frequently overwhelmed. With AIMD only a few packets were dropped, but many packets are dropped with AIAD.



As shown in the above graph, AIAD never allows the single flow to use the full 1 Mbps bandwidth. Instead, the bandwidth oscillates between 928 and 968 kbps with no evidence that it will ever converge.

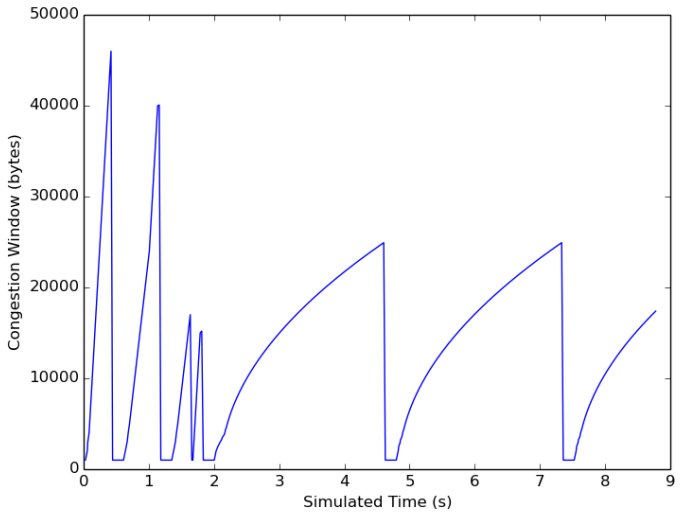
When AIAD is used instead of AIMD, TCP ceases to show stable behavior. The bandwidth is not fully utilised, and the congestion window does not really help to prevent network congestion.

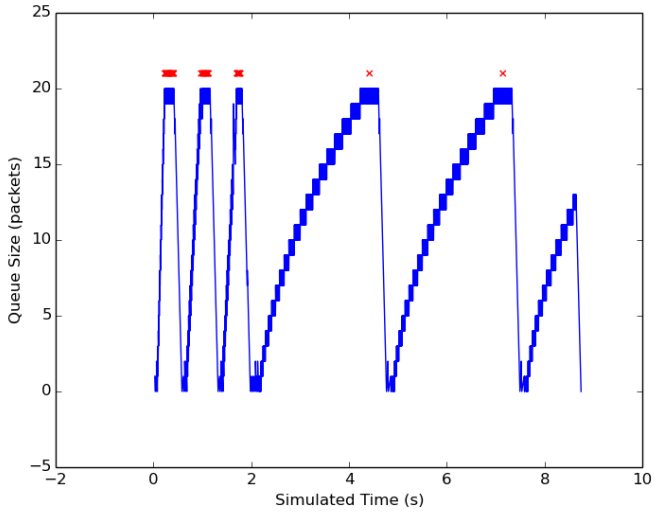
3.2 AIMD - Different Multiplicative Constant

I changed my `tcp.py` code to use AIAD with a multiplicative constant of 5/6 instead of 1/2:

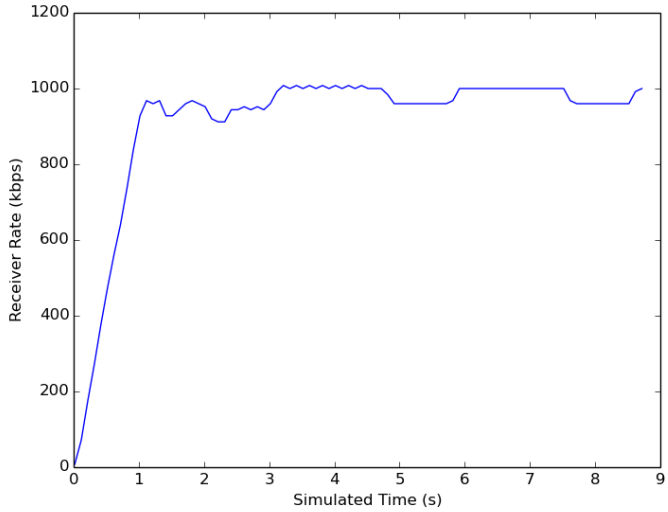
```
def do_retransmit(self):
        # Set ssthresh and cwnd
        #self.ssthresh = max(self.cwnd/2, self.mss)
        self.ssthresh = max(self.cwnd * 5 / 6, self.mss)
        self.cwnd = self.mss
```

I sent a 1MB file over a single flow.





With a multiplicative constant of 1/2, the congestion window started showing the familiar sawtooth pattern after about 1 second. With the new constant of 5/6, the congestion window takes a little longer before starting the sawtooth pattern. As shown in the Queue Size graph above, this leads to more packet loss in the first 2 seconds, but it does start to regulate itself better after that.



With a multiplicative constant of 5/6, it takes the flow a little longer before it uses the full bandwidth amount, but it does get there eventually. Using a multiplicative constant of 5/6 causes the transfer time to be about 12% longer than the transfer time is with a multiplicative constant of 1/2.

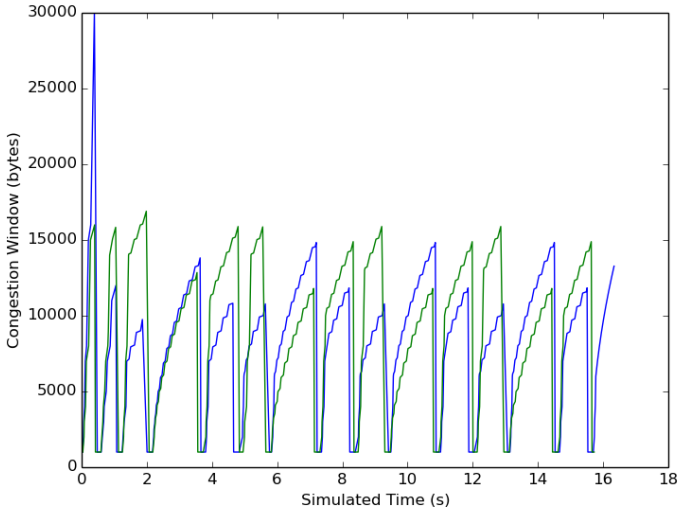
When a different multiplicative constant is used, TCP still shows stable behavior, though for this particular case it seems that 1/2 is a better constant than 5/6 is.

3.3 Competing AIMD

I changed my `tcp.py` code to use different multiplicative constants for a two-flow scenario:

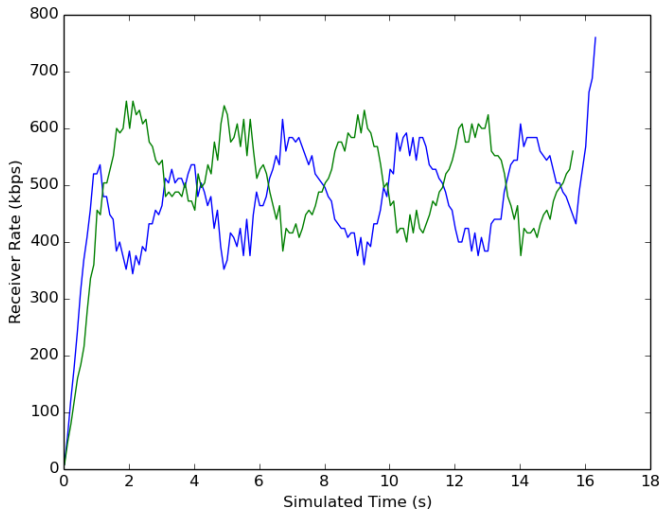
```
1 def do_retransmit(self):
2         # Set ssthresh and cwnd
3         if self.source_port == 1:
4                 self.ssthresh = max(self.cwnd/2, self.mss)
5         else:
6                 self.ssthresh = max(self.cwnd * 5 / 6, self.mss)
7         self.cwnd = self.mss
```

A 1 MB file is sent over both flows.



When both flows had the same multiplicative constant of 1/2, they eventually got to the point where their congestion windows overlapped almost perfectly. In this case, the second flow (green line) with a

constant of 5/6 usually has a larger congestion window than the first flow does.



Both flows still seem to get their fair share of the bandwidth, but it is different from the case where both flows use a constant of 1/2. In the first case, both flows seemed to oscillate back and forth between 450 and 550 kbps. In the second case, the flows seem to oscillate back and forth between 350 and 650 kbps. In both cases, each flow averages about 500 kbps for the file transfer, but in the second case the distance from the average is more extreme for each flow.

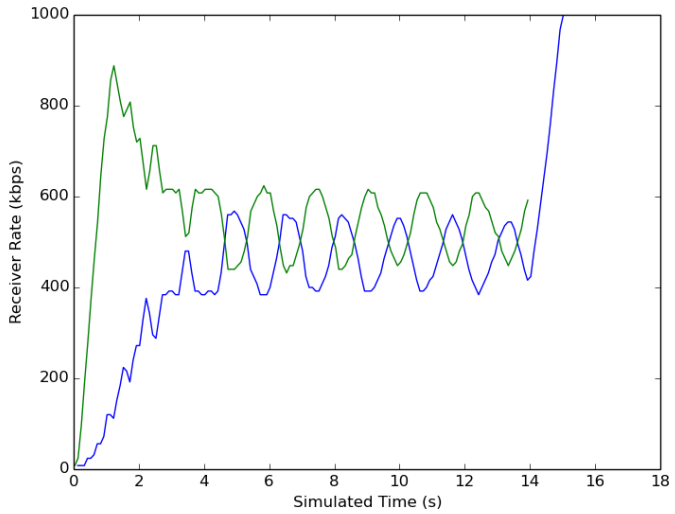
3.4 Competing RTT

I created a new network, `split-network.txt`, which is set up like this:

```
# n1
#   \
#    n3 -- n4
#   /
# n2
n1 n3
n2 n3
n3 n1 n2 n4
n4 n3

# link configuration
n1 n3 1Mbps 100ms
n3 n1 1Mbps 100ms
n2 n3 1Mbps 10ms
n3 n2 1Mbps 10ms
n3 n4 1Mbps 10ms
n4 n3 1Mbps 10ms
```

I wrote a new python script `compete_rtt.py` and a new bash script `run_compete_rtt.sh` that use `split-network`. One flow is routed betwen n1 and n4, the other between n2 and n4. I send a 1 MB file over each flow, starting at the same time.



The first flow (blue line) has a longer propagation delay on its first link than the second flow (green line) has on its first link. The first flow's `cwnd` does not grow as fast because the round trip time is longer, and the `cwnd` only grows when it receives ACKs. Due to this, the first flow starts off slower and the second flow uses more bandwidth. Once the first flow catches up, the two oscillate back and forth around the equal share line. Each flow still gets its fair share of bandwidth.