John Dai, Sajal Kantha

CS 356

Benson/Yang

Lab 4: Congestion Control

Protocol Design

First we started with our base code from Lab 1. We had to make sure the connection closes itself after the sender finishes transferring a file. This was relatively simple since our Lab 1 code handled most of it already, we just had to add an if-statement to allow the receiver to automatically send an EOF during initialization. This works even if the sender is not running yet since the receiver’s sender buffer will timeout so it will keep sending until the sender ACKs it, at which point half the conditions for closing a connection has been met.

Next, we had to improve our TCP protocol to improve throughout. Through much trial-and-error we feel that we have come up with a protocol that consistently outperforms the provided measuring tool running on the VM, and it even reaches peak throughput of over 10Mb/sec and can average over 7Mb/sec. The main idea was to keep the pipe full with useful packets without congesting the network. First we wanted to implement simple flow control by having the sender send its maximum window size in the *rwnd* field of the packet (i.e. 200). The sender will parse this value from the ACKs it receives and will make sure its window size never exceeds this. This prevents the sender from reaching a state that guarantees loss which would lead to useless packets congesting the network.

Then we started looking at increasing the output of the sender. For this we modeled our protocol on TCP Reno and RFC2581. When it starts up we have our version Slow Start implemented, meaning that for each ACK the sender receives that is greater than *Last ACK Received* it increases the sender window by 1. This makes for exponential probing of the link’s capacity. Then when packets inevitably start timing out using RTO due to congestion, instead of setting the window size to 1 the sender changes into a “transition” state. In this “transition” state, it continues resending packets that have timed out but doesn’t send new packets or change the window size. Once all the packets in the sender buffer are ACKed, then the sender changes into the AIMD state. Here it halves the window size and starts sending new packets again. Also, at every RTT it increases the window size by 1 using *rel\_timer*. Again, if a timeout occurs it will go back into the “transition” state, halve the window size, and then go back to AIMD.

This leads to the change we made in calculating RTO and RTT. For this we basically followed RFC2988 very closely but made sure that RTO never fell below 4 times the value of the provided propagation delay (20ms). This means that we will sometimes have more generous estimates for RTO which compensates for the Relayer’s queue, avoiding retransmission for some packets that were just simply slower.

A design alternative we tried was to also implement Fast Retransmit and Fast Recovery. However, we found that when packets are dropped and not ACKed, they tend to do so consecutively around 10 or so packets at a time. Intuitively, Fast Retransmit would be slow in this case, so we would have to implement the modified version in RFC2582. We found through testing that our throughput up to this point had been sufficient to pass the 80% requirement so we decided to leave NewReno’s Fast Recovery for later if we needed.

Finally we also had to directly measure a couple performance metrics. For the total transit time we decided to measure the time between the sender’s first packet was sent to when the sender received back the ACK for its last EOF. To implement this we stored the current system time in microseconds as a packet is sent if the *Last Frame Sent* equals 1, and we output the difference between the current system time and this stored time when an ACK corresponding to the sequence number of the EOF is received.

Then for intermediate values of throughput, we had a variable for the sender that stores the number of bytes sent (including duplicates) and a variable for the receiver that stores the number of bytes received (not including duplicates). This value would be accessed every 20 calls of *rel\_timer* (every 200ms), converted to throughput, outputted to a uniquely named space-delimited text file, and reset to 0. Clearly the throughput values for the receiver will be more accurate since it does not double count packets already seen. For intermediate values of the sender congestion window we simply outputted the sender window size every 200ms to the same space-delimited text file. The other metrics such as average throughput and Jain’s Fairness Index would be calculated by hand later using our collected data.

Analysis

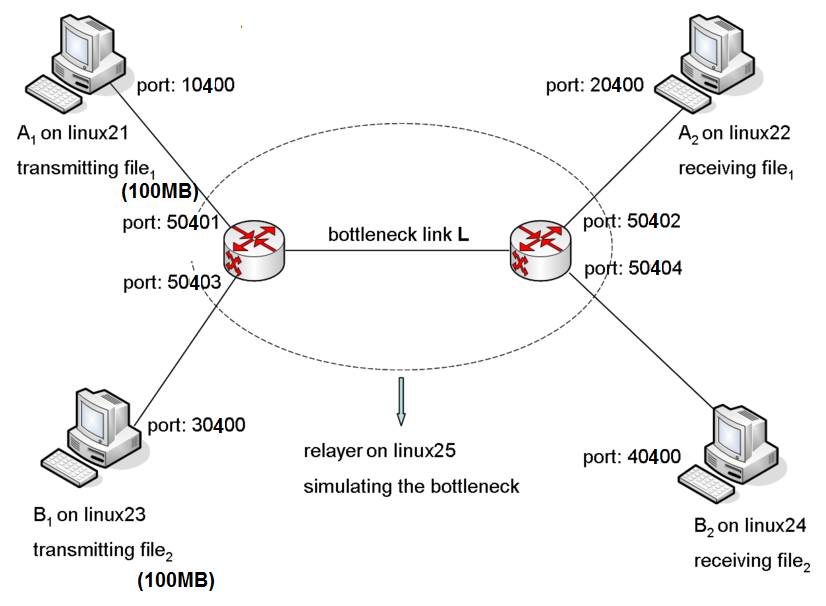
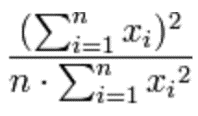


Figure 1: Our testing topology

Measured throughput on our VM using the provided measuring tool: 3.7mb/sec

Measured throughput on the Linux clusters using the provided measuring tool: 3.0 to 5.5mb/sec

Equation for average throughput: 100(MB) \* 8(b/B) / Transit time(sec)



Equation for Jain’s Fairness Index: for *n* senders each with average throughput *x­i*

For all the following graphs the Y-axis measures throughput in bits per second or window size in units of MSS, and the X-axis measures the current system time in milliseconds. The series labels are in the top right corner and describe the number of pairs of links, whether it represents the sender or receiver, the trial number, and the link pair number. As Shown in Figure 1 we send a 100MB file in all cases. All are plotted using Gnuplot. For example, for the first graph: “plot ‘1link\_senderstats1.dat’ w lines, ‘1link\_receiverstats1.dat’ w lines”. The “using 1:3” tag in the series labels means the y-values represent window size.

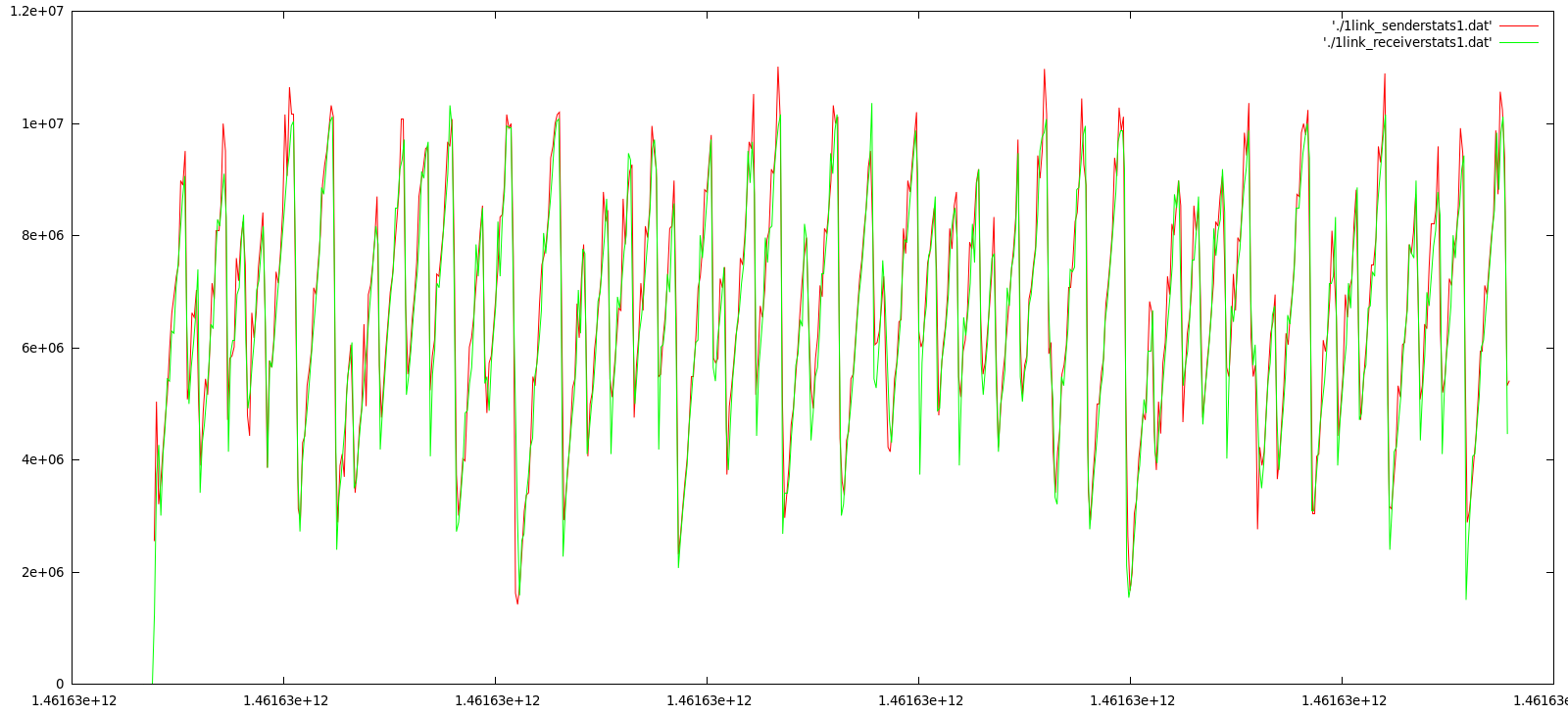
As stated in the Protocol Design, the sender and receiver throughputs are different because the sender side measures the total data (1016 bytes per packet) sent, while the receiver measures the total data (1016 bytes per packet) received from packets that **had not been seen before**. Therefore we use the throughput measured by the receiver as a more accurate measurement.

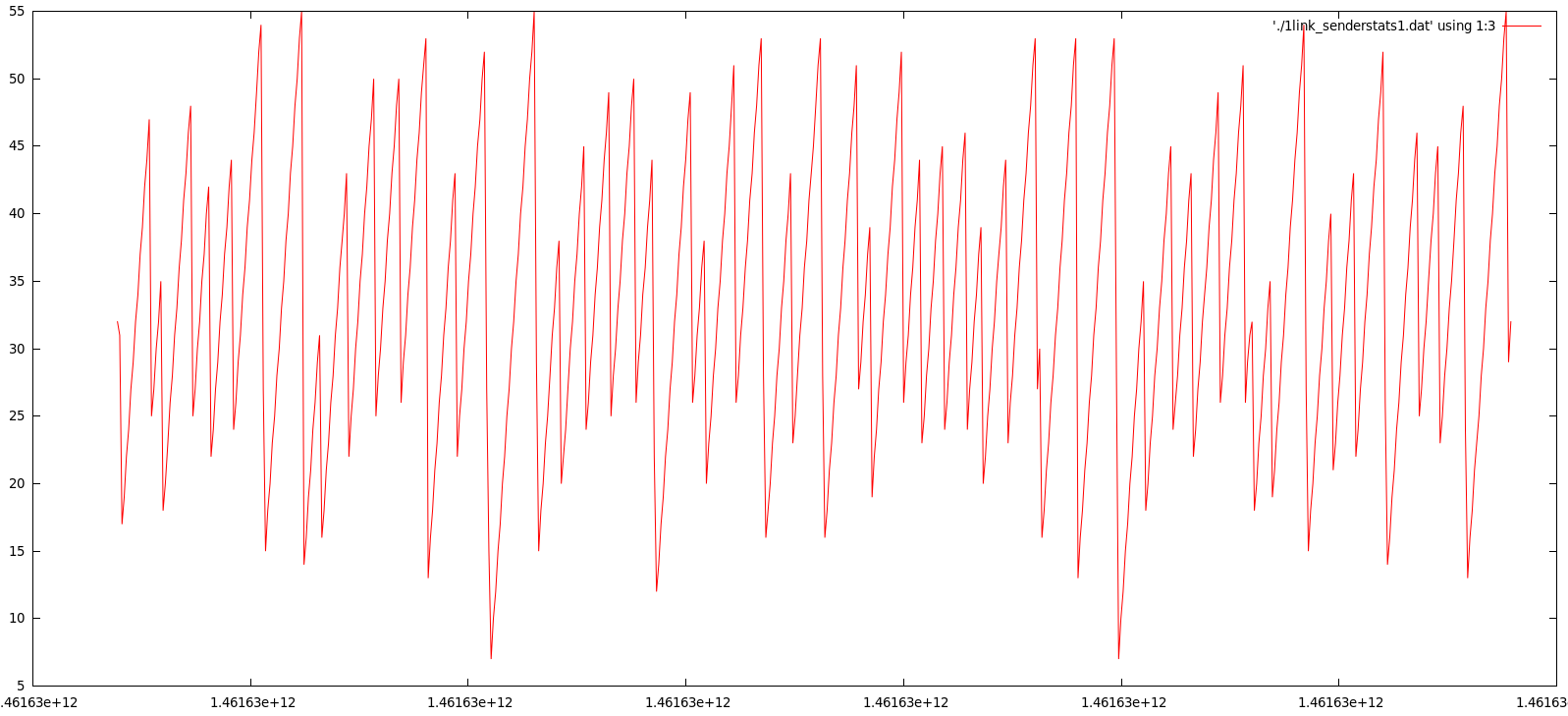
**1 pair of links, 100MB file, trial 1:**

Transit time = 128.397888 sec

Average throughput = 6.23 mb/sec

X-axis is about 20 seconds per tick.



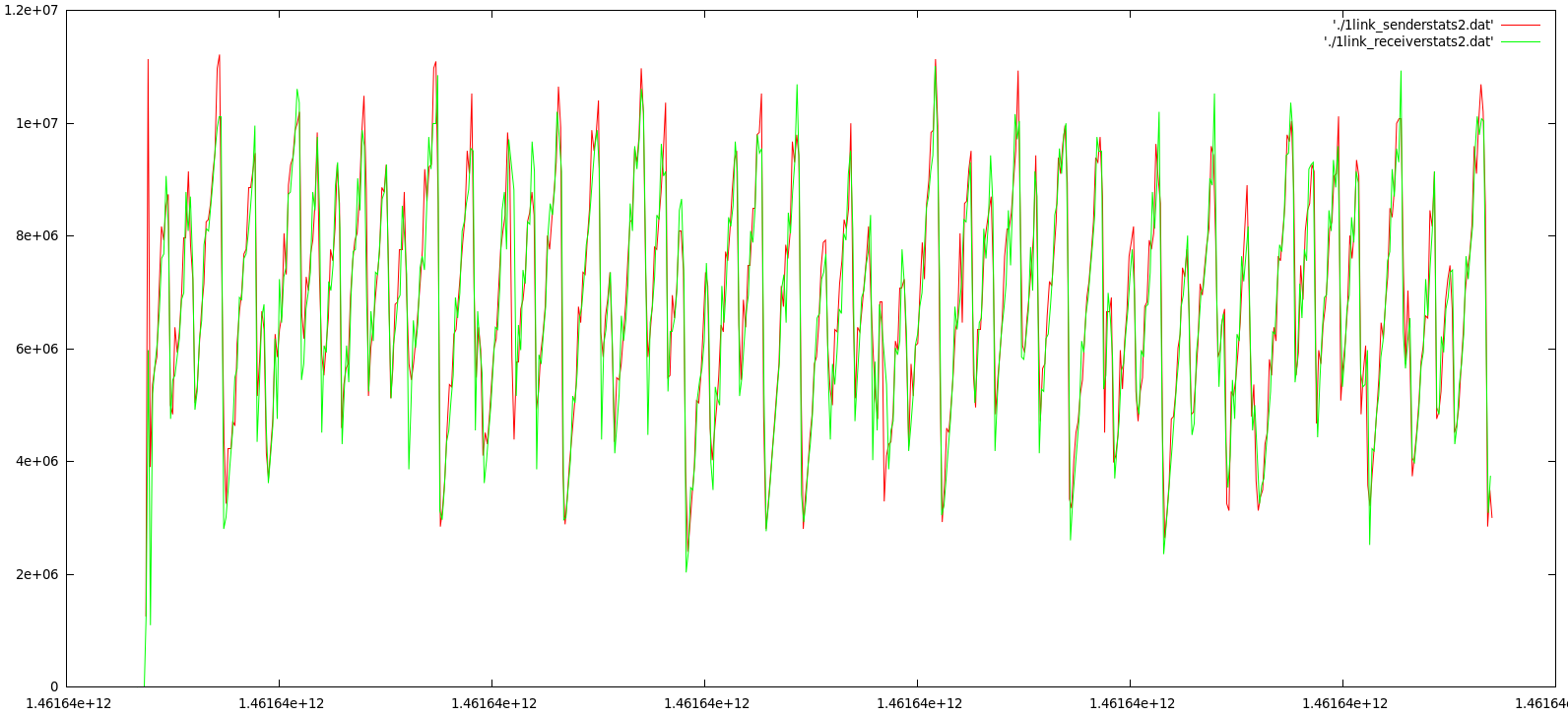


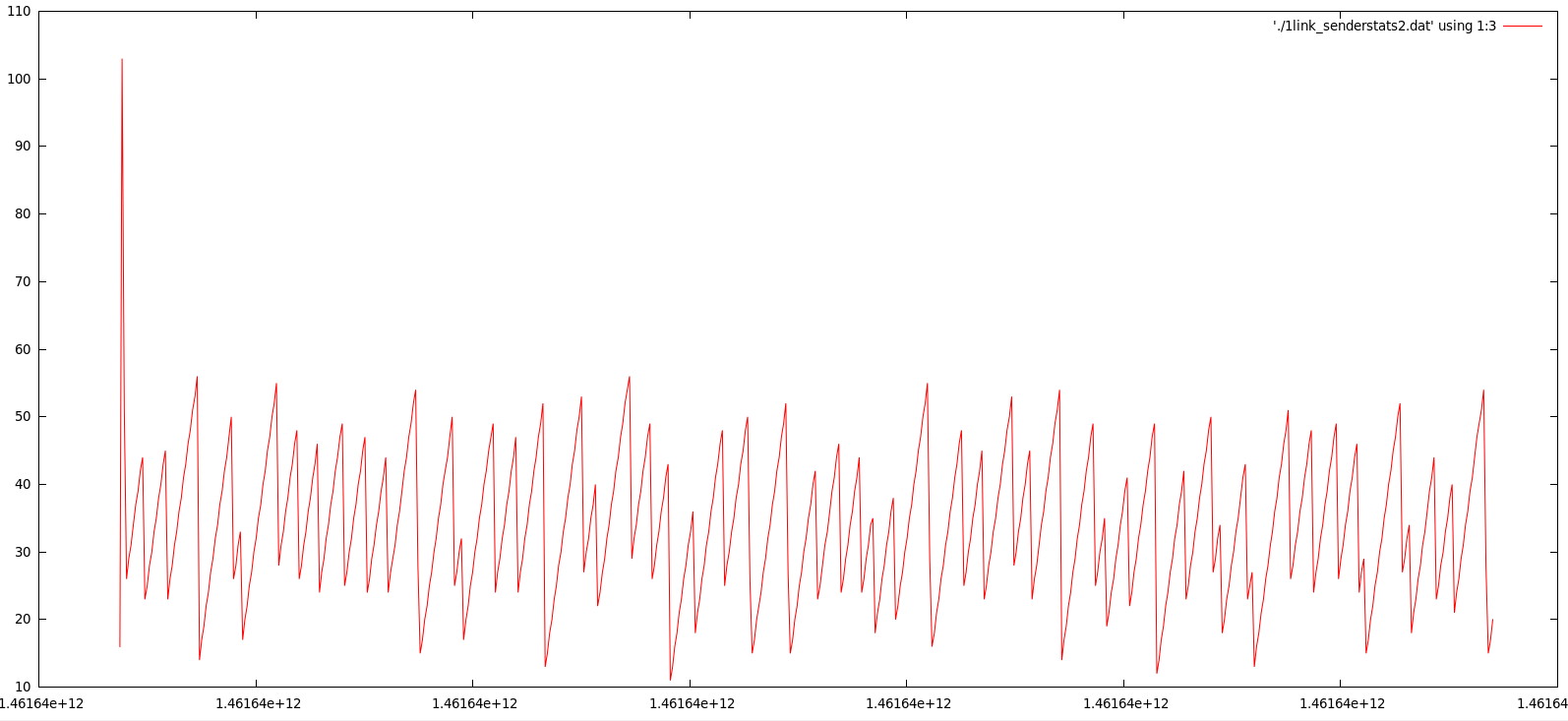
**1 pair of links, 100MB file, trial 2:**

Transit time = 126.904672 sec

Average throughput = 6.30 mb/sec

X-axis is about 20 seconds per tick.





From these single sender-receiver pair trials our protocol seems to be doing well. Average throughput is about **twice** the value of what was measured using the measurement tool. We see peak throughput of over 10mb/sec which seems to be the maximum possible value.

Some interesting observations:

* The throughput and window size exhibit a saw tooth pattern which is very similar to the TCP behavior shown in lecture
* The window size and sender throughput is very high in the beginning due to Slow Start, and then it quickly decreases and converges to a steady-state oscillation in AIMD.
* The throughput of the sender at its highest peaks (in red) exceed the throughput of the receiver (in green) by a sizeable margin probably since at these high levels of output congestion is higher so the receiver cannot keep up and might also see duplicate packets.

**2 pairs of links, 2 100MB files, trial 1:**

Transit time for 1st sender = 194.205483 sec

Transit time for 2nd sender = 255.139440 sec

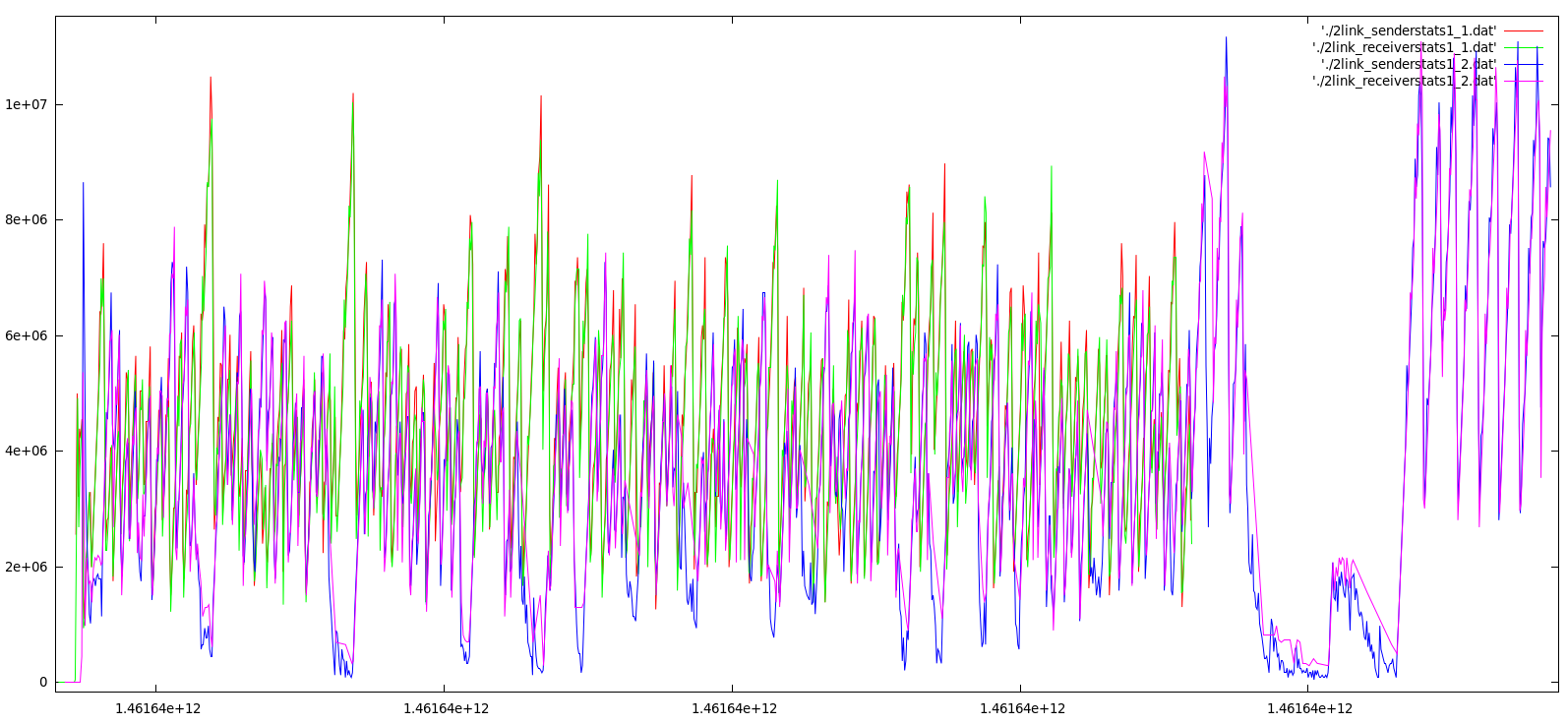
Average throughput for 1st sender = 4.12 mb/sec

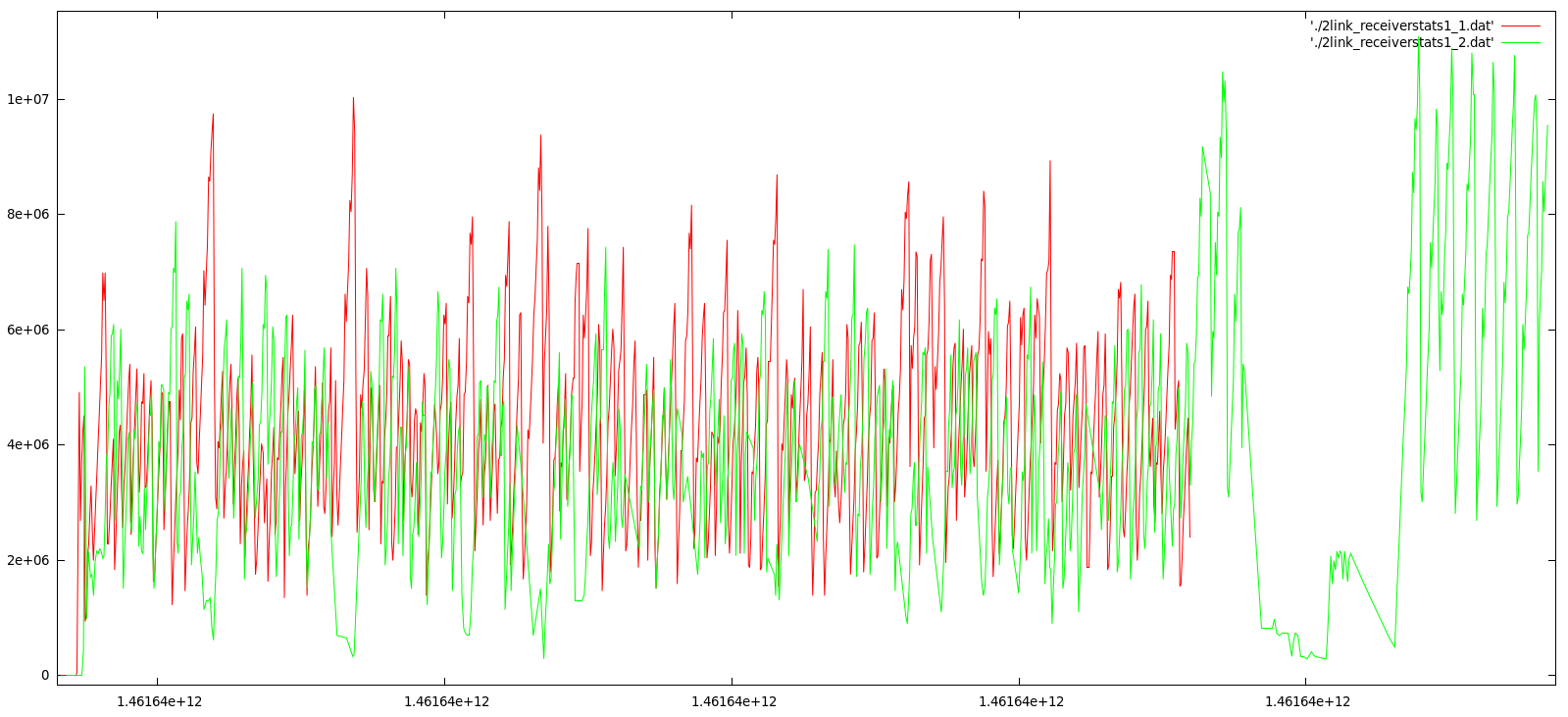
Average throughput for 2nd sender = 3.14 mb/sec

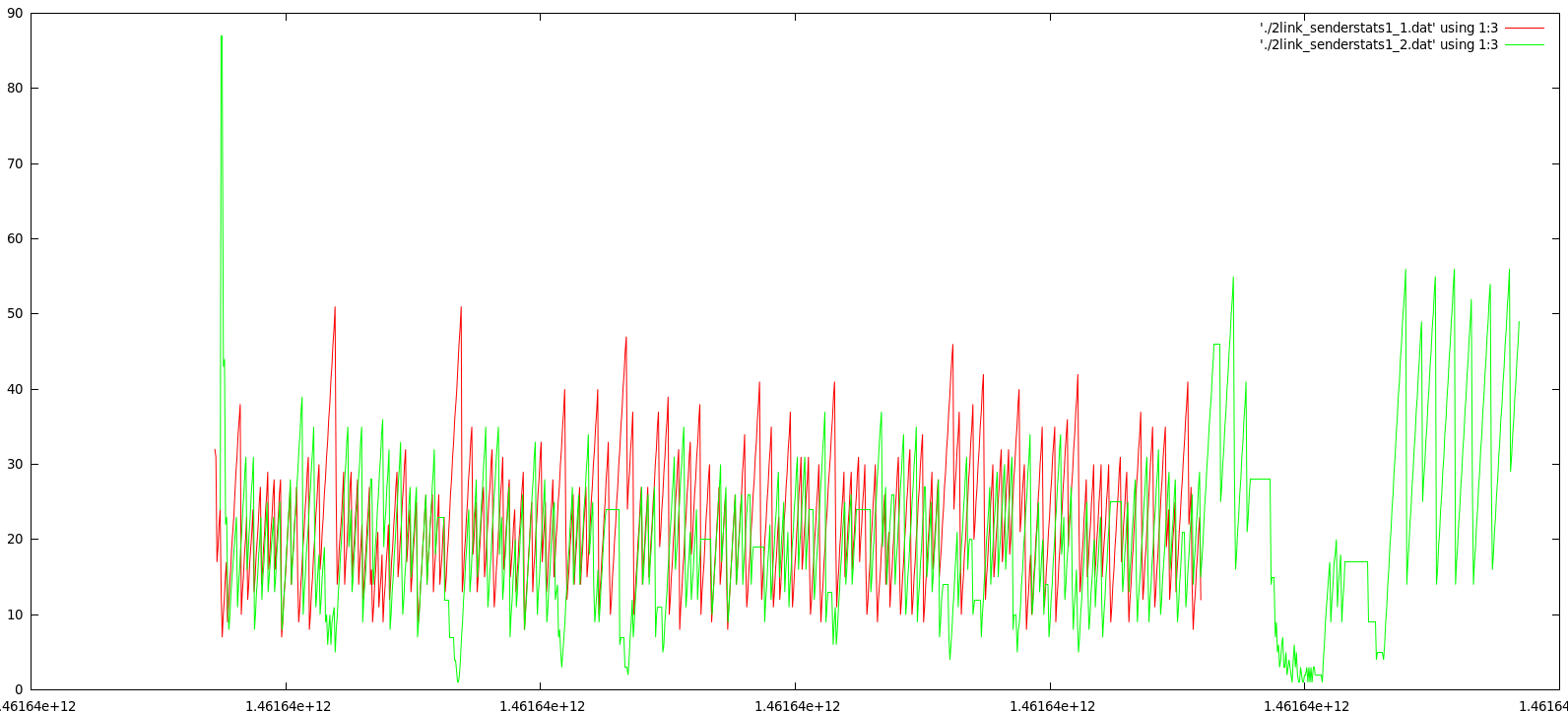
Total average bandwidth = 7.26mb/sec

Jain’s Fairness Index = 0.982

X-axis is about 50 seconds per tick.







**2 pairs of links, 2 100MB files, trial 2:**

Transit time for 1st sender = 184.454704 sec

Transit time for 2nd sender = 231.588672 sec

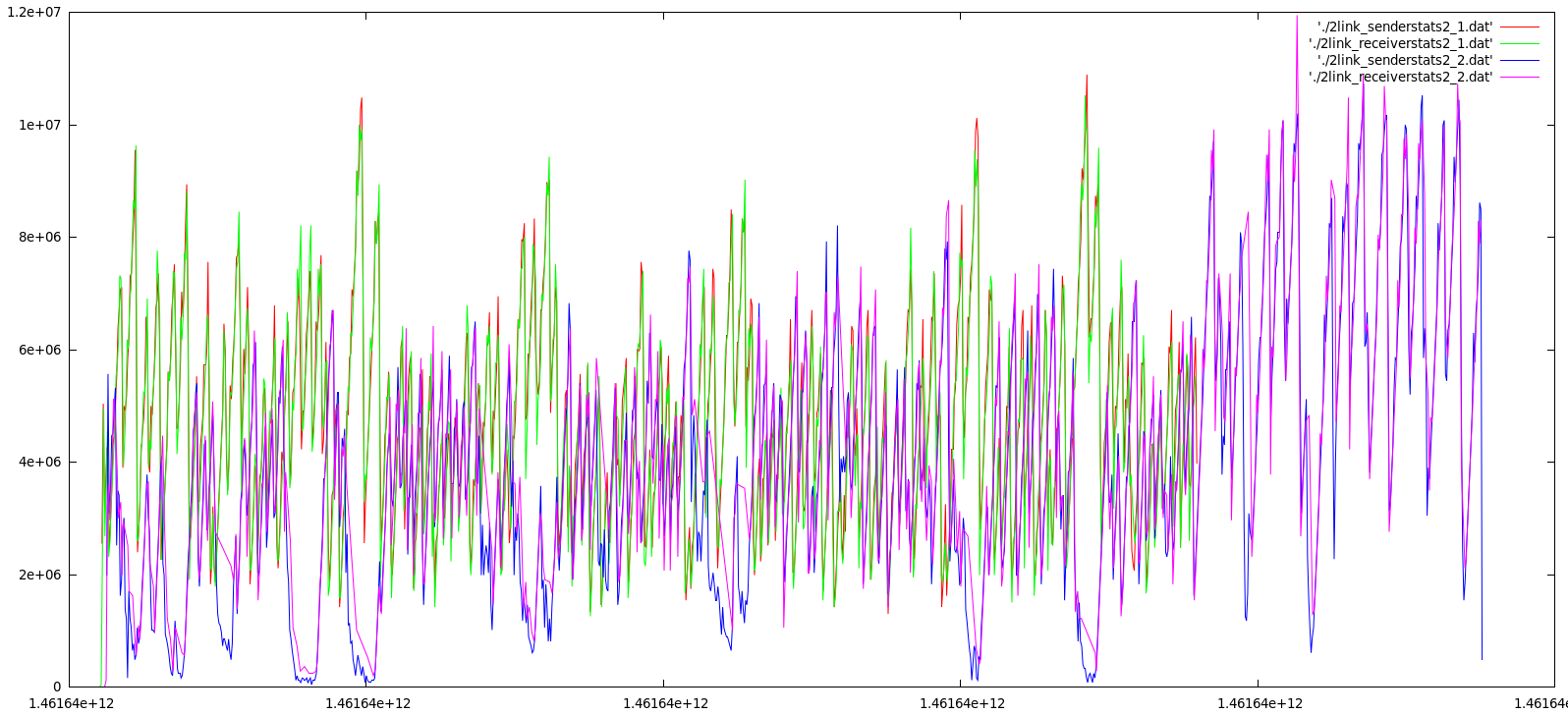
Average throughput for 1st sender = 4.34 mb/sec

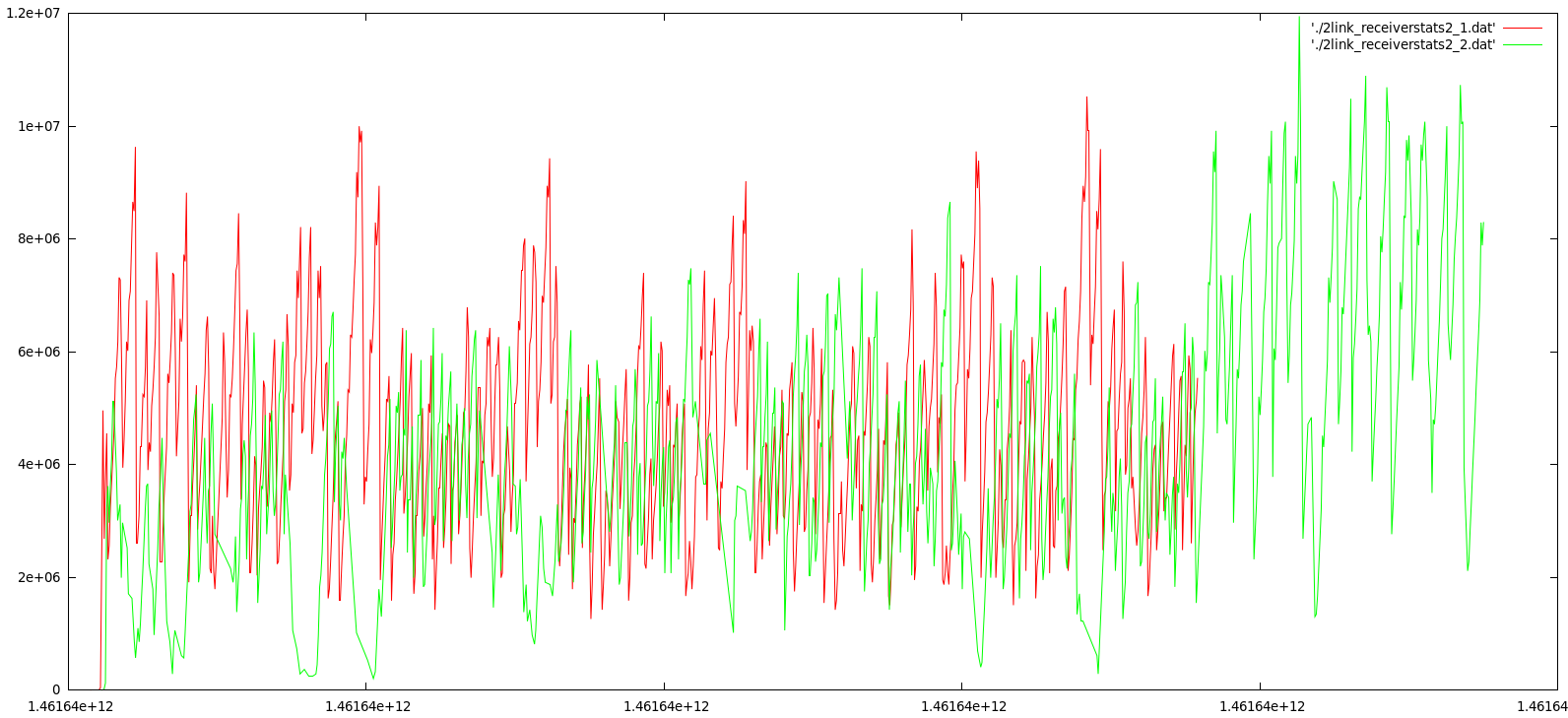
Average throughput for 2nd sender = 3.45 mb/sec

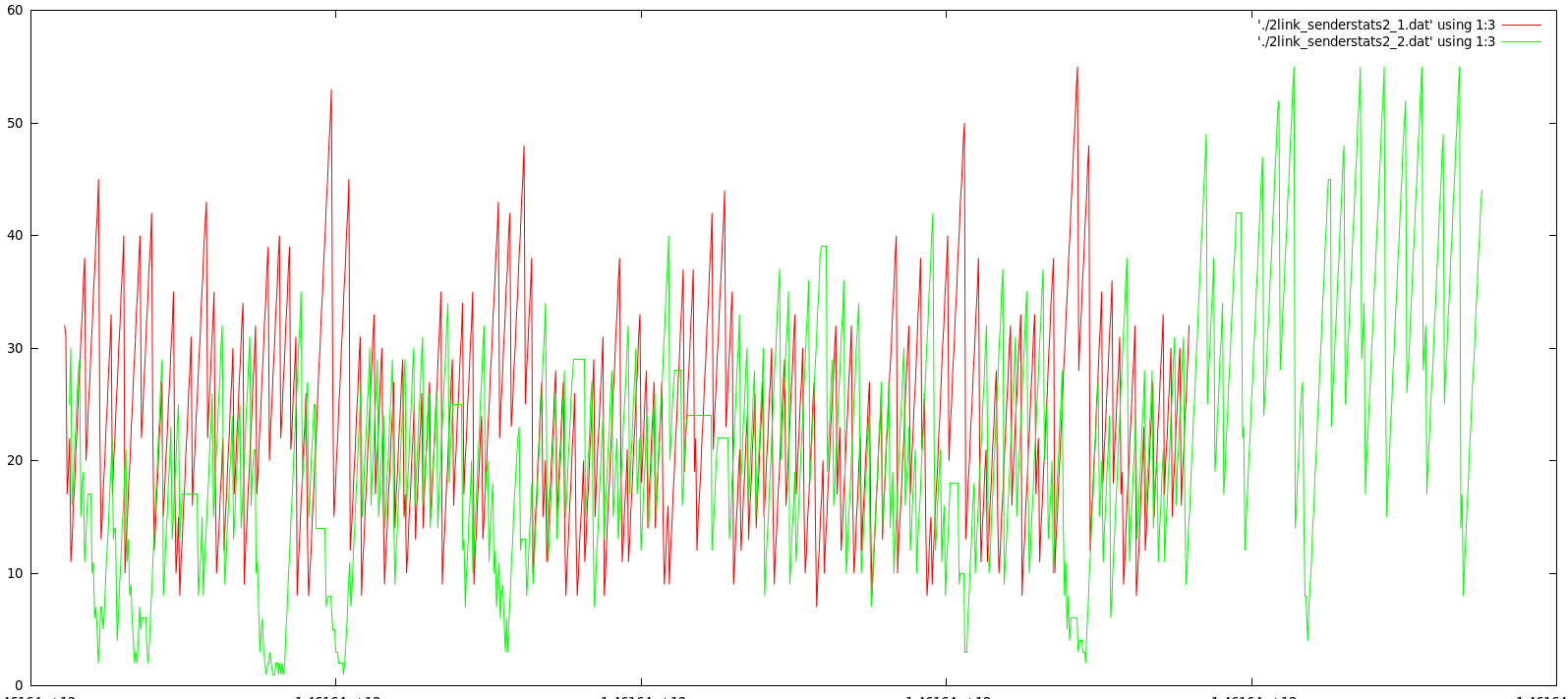
Total average bandwidth = 7.79mb/sec

Jain’s Fairness Index = 0.987

X-axis is about 50 seconds per tick.







From these double sender-receiver pair trials our protocol also seems to be doing well. Average throughput is also about **twice** the value of what was measured using the measurement tool. We still see peak throughput of around 10mb/sec, although this seems to happen less often. Our fairness values are also very close to **1**. However, based on the graphs we see that the sender that starts later has a bit of a disadvantage in throughput and window size.

Some interesting observations:

* The overall oscillation and saw tooth pattern still seem to be present, but now it is noticeably more chaotic and random. This could be explained by the competition of resources between the two senders and heavy CPU loads.
* There are times when the first sender peaks up at the same time that the second sender peaks down. This more or less preserves overall bandwidth but may be less fair to the second sender.