Advanced Networking 2018

Lab #1: TCP Congestion Control

Report

GROUP: 1

Authors:
Rick van Gorp, rick.vangorp@os3.nl
Luc Gommans, os3@lucgommans.nl
University of Amsterdam

Q1.1 Plot a graph showing CWND versus time from 0.0s to 100.0s.

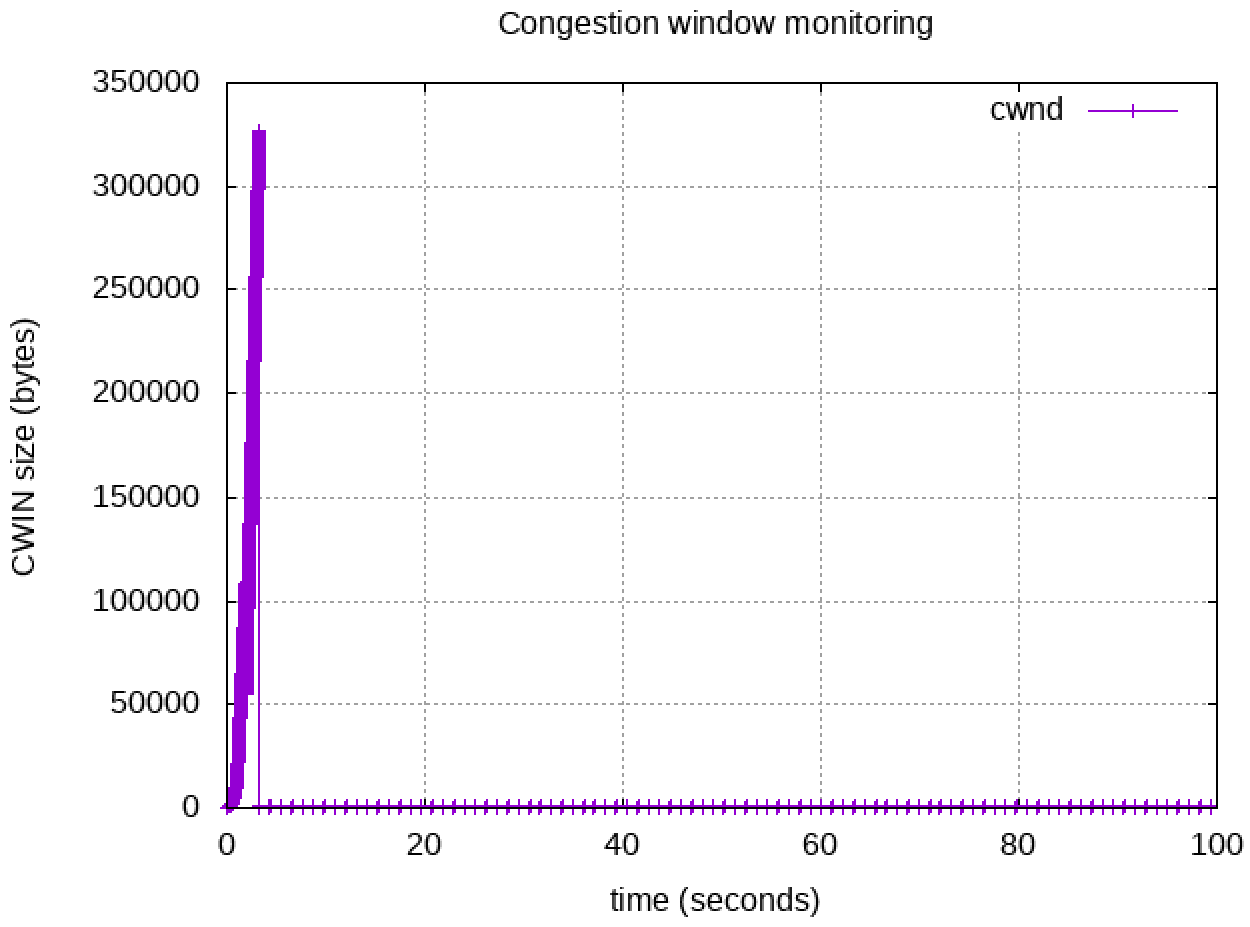


Figure 1: T1Q1 CWND from 0 to 100 seconds

Q1.2 Plot a graph showing SSTH versus time from 0.0s to 100.0s.

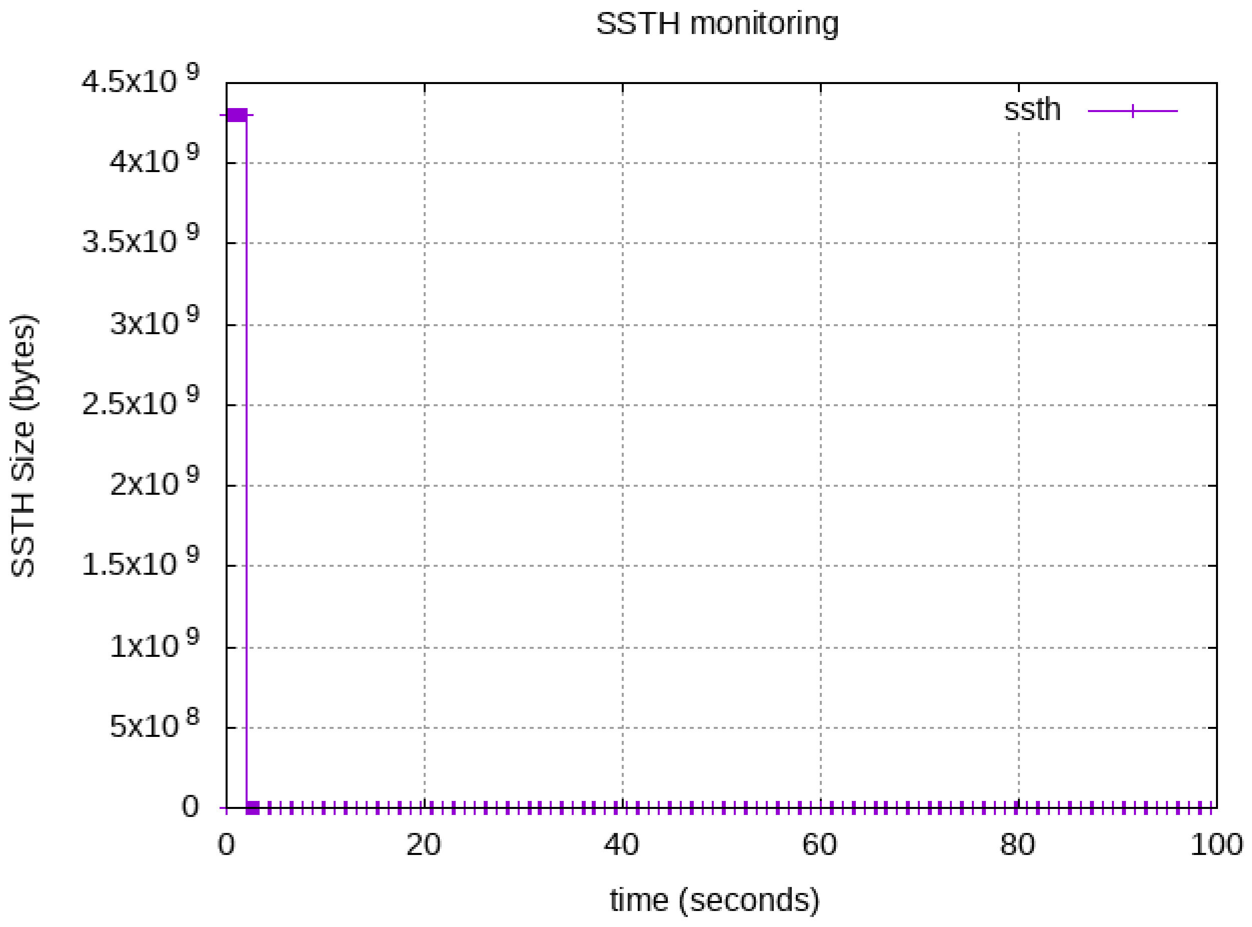


Figure 2: T1Q2 SSTH from 0 to 100 seconds

Q1.3 Find the points where the slow-start, congestion-avoidance, fast retransmit/fast recovery states begin.

Table 1 shows the points where the slow-start, congestion-avoidance and fast recovery states begin. The results are based on the reno datafiles generated in the vagrant boxes and the states described in the slidedeck of lecture 1. We found the slow-start state starts at 0 seconds and is followed by the fast recovery state as $ssthresh$ halves and $cwnd$ becomes 3 higher than $ssthresh$. After the fast recovery state we see that the algorithm goes to the slow start state again and then, after some time $cwnd$ exceeds $ssthresh$. From here the transition from slow start to congestion-avoidance occured.

Table 1: Points of state transitions NewReno algorithm

Time (s)	Current CWND (bytes)	New CWND (bytes)	New State	Event
0.00000	0	340	slow-start	start
1.93189	109 480	55 590	fast-recovery	dupACKcount==3
3.26916	326 570	340	slow-start	timeout
3.30286	340	680	congestion-avoidance	cwnd>=ssthtresh

Q1.4 Plot a graph showing CWND versus time from 0.0s to 100.0s.

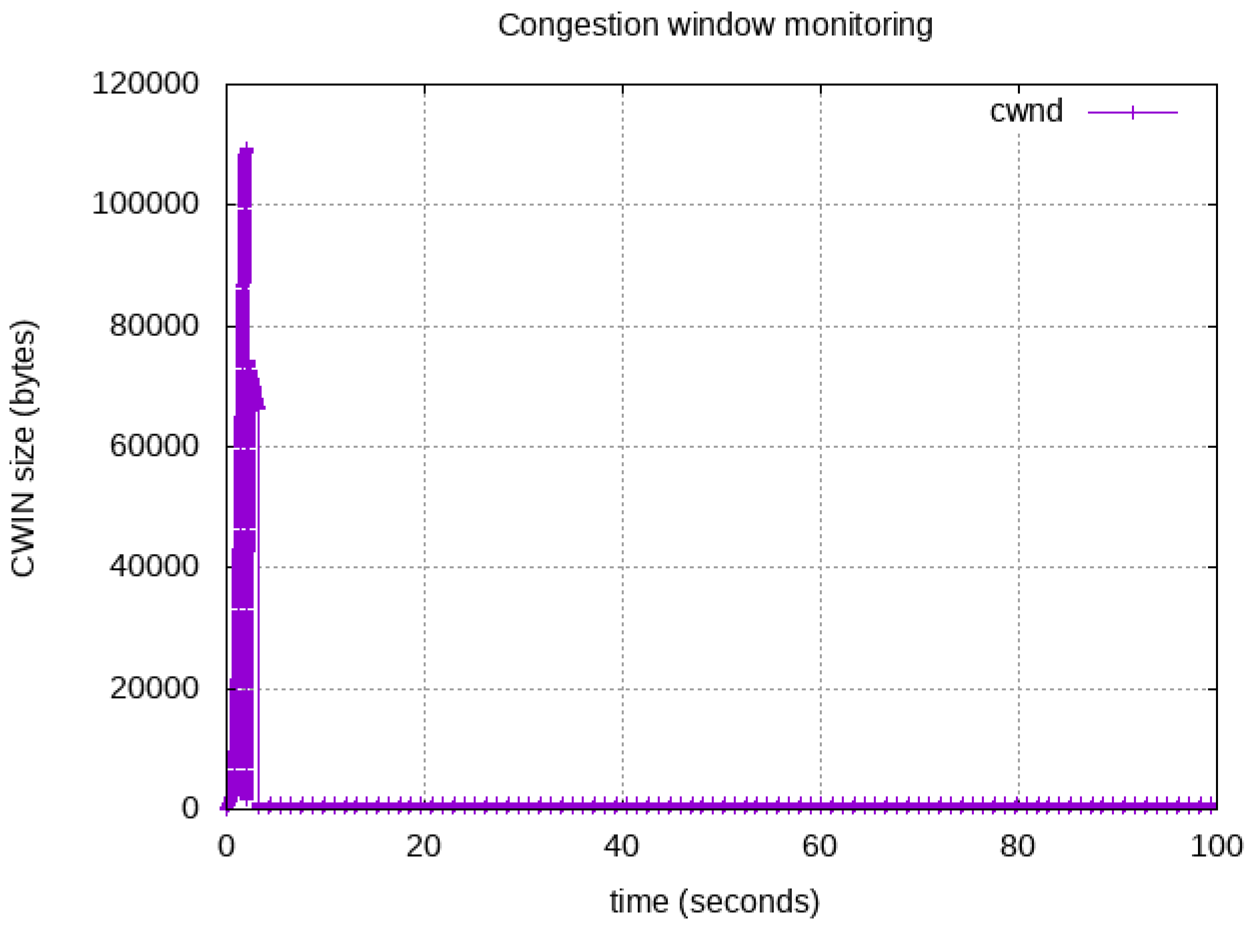


Figure 3: T1Q4 CWND from 0 to 100 seconds

Q1.5 Plot a graph showing SSTH versus time from 0.0s to 100.0s.

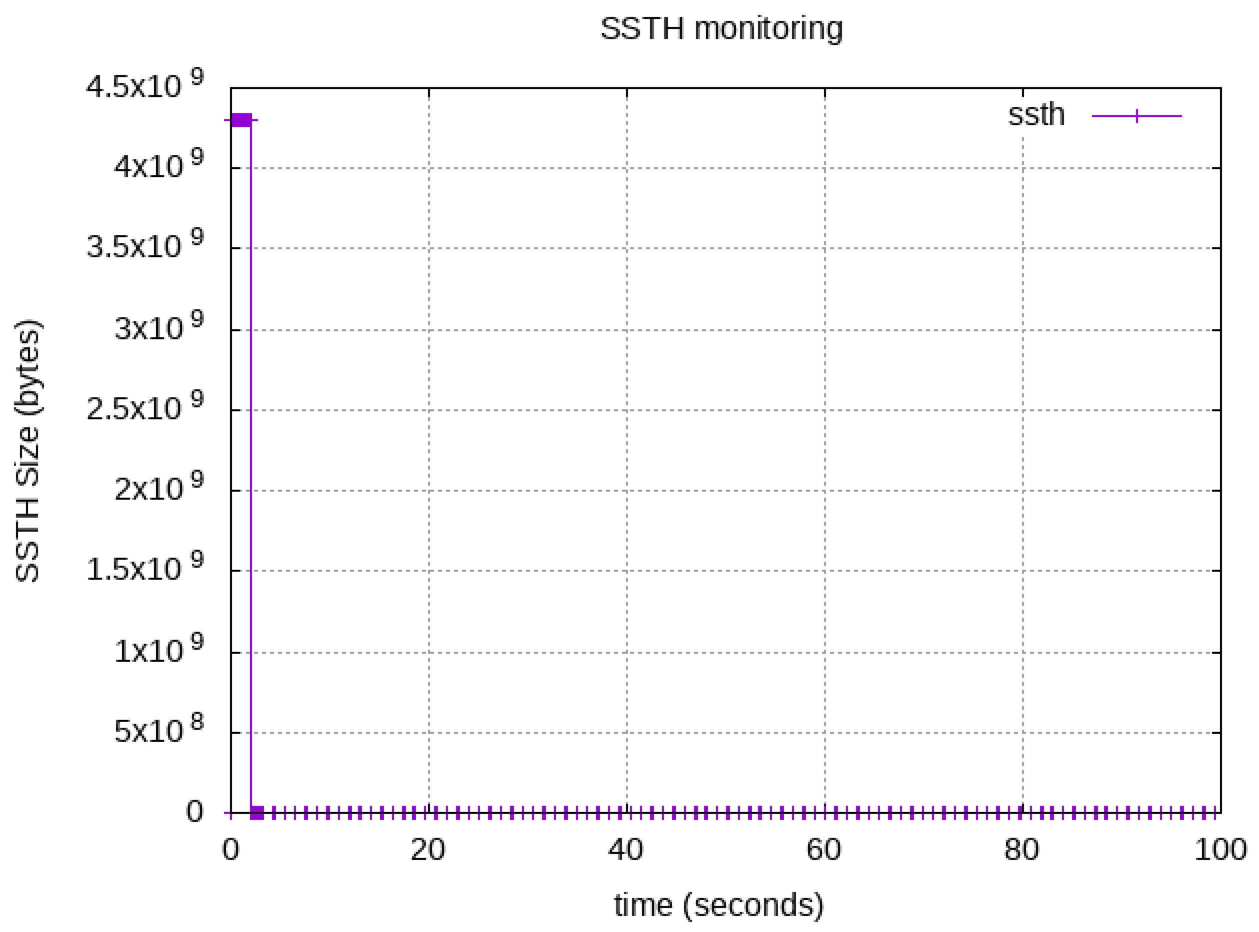


Figure 4: T1Q5 SSTH from 0 to 100 seconds

Q1.6 Find the points where the slow-start, congestion-avoidance, fast retransmit/fast recovery states begin.

Table 2: Points of state transitions WestWood algorithm

Time (s)	Current CWND (bytes)	New CWND (bytes)	New State	Event
0.00000	0	340	slow-start	start
1.93189	109480	1700	fast-recovery	dupACKcount==3
2.26758	74 120	73 100	congestion avoidance	New ACK
3.19086	66 640	340	fast-recovery	dupACKcount==3

The strange part of this graph is the section between 2.2 and 3.3 seconds, where the congestion window slowly decreases. We could initially not attribute it to the state machine, nor could we find a paper which described this behavior. Looking in the source code of NS3, we believe the following line is responsible for it:

```
m_currentBW = m_ackedSegments * tcb->m_segmentSize / rtt.GetSeconds ();
```

Because the estimated RTT increases, the current bandwidth estimate decreases from 11 301 to 3081. All other variables in the source code appear to be increasing, therefore it must be this declaration.

Q1.7 Discuss and motivate the differences you observe between the NewReno and this algorithm.

For NewReno we see that fast recovery occurs within a shorter timeframe compared to TCPWestWood. TCP-NewReno however, appears to have a higher initial peak. It appears that TCPWestwood acts differently in the congestion avoidance phase as we see the window slightly decreasing. Both algorithms show a time-out afterwards. After the sizes have been recovered again, the algorithms show the same behavior. The maximum peak of $cwnd$ is shown at 850 bytes and the minimum peak of $cwnd$ is shown at 340 bytes.

Q2.1 Plot a graph showing the CWND and ssthresh versus time with all the data you get. These two metrics are in one graph.

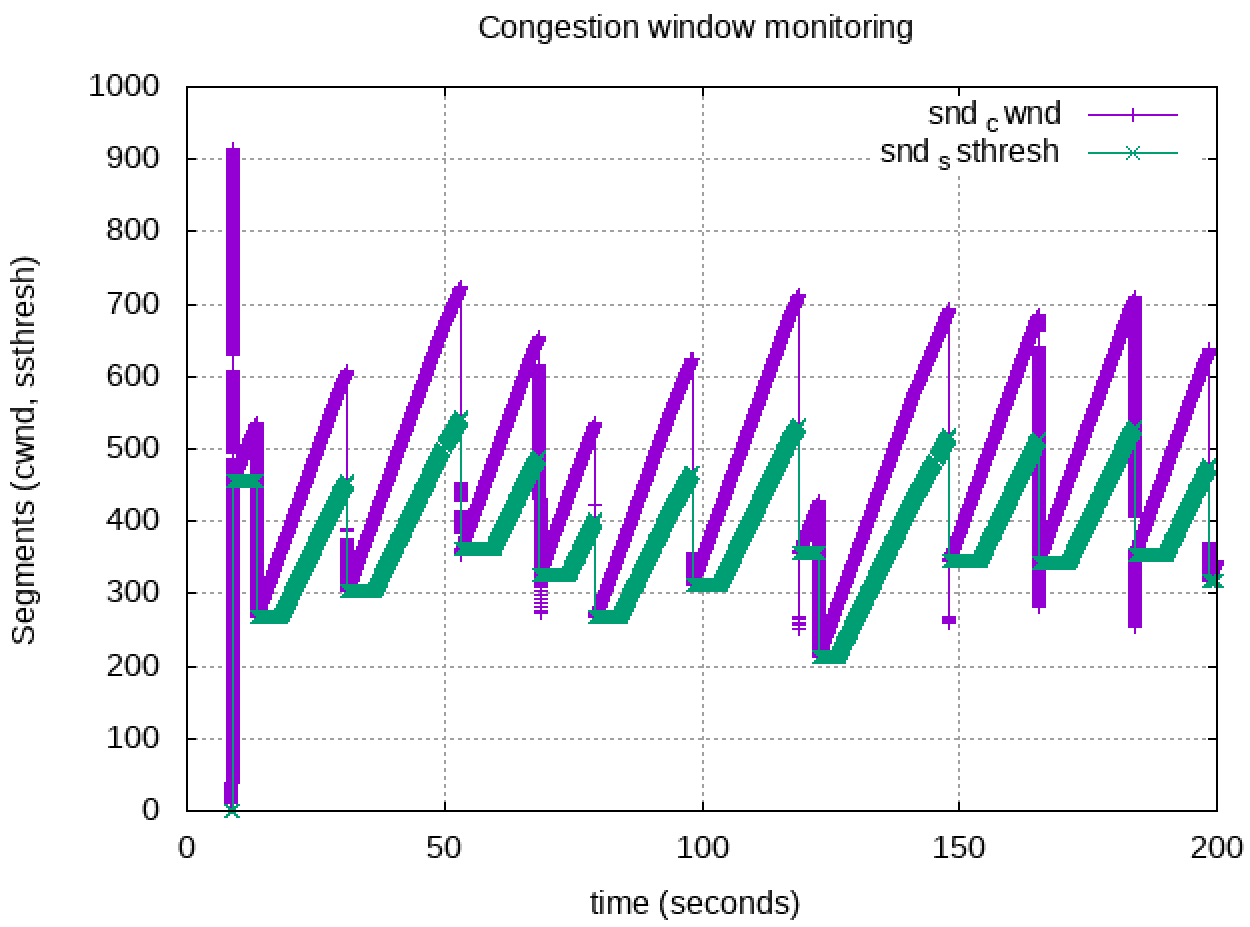


Figure 5: T2Q1 CWND and ssthresh 0 to 200 seconds

Q2.2 Briefly discuss the changing process.

We see that it initally starts in slow start, until the source received three duplicate ACKs. It then transitions to fast recovery, where it slowly increases the congestion window until it receives a new ACK. We are now in congestion avoidance. From there we see multiple transitions between congestion avoidance and fast recovery, as we see the threshold halving and the CWND is adjusted to `ssthresh + 3MSS`.

Q2.3 Plot a graph showing CWND versus time with all the data you get.

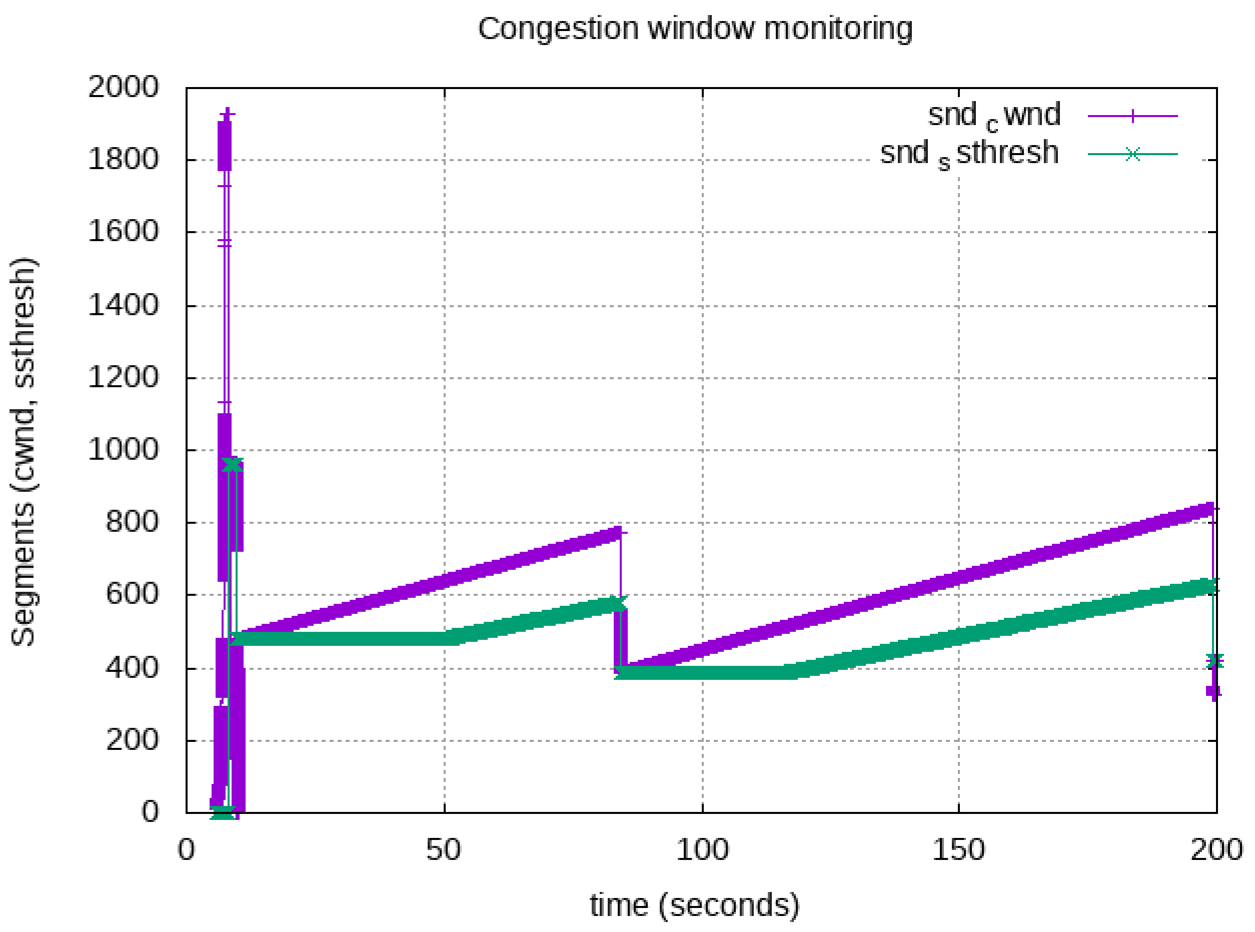


Figure 6: T2Q3 CWND and ssthresh 0 to 200 seconds

Q2.4 Compare this graph with the one from

Both graphs start similarly. The relatively fast, oscillating changes between the fast recovery and congestion avoidance states, are much slower in the latter (figure). The congestion avoidance states in figure hold for a longer time than in figure .

Q2.5 Plot a graph showing CWND and ssthresh versus time with all the data you get.

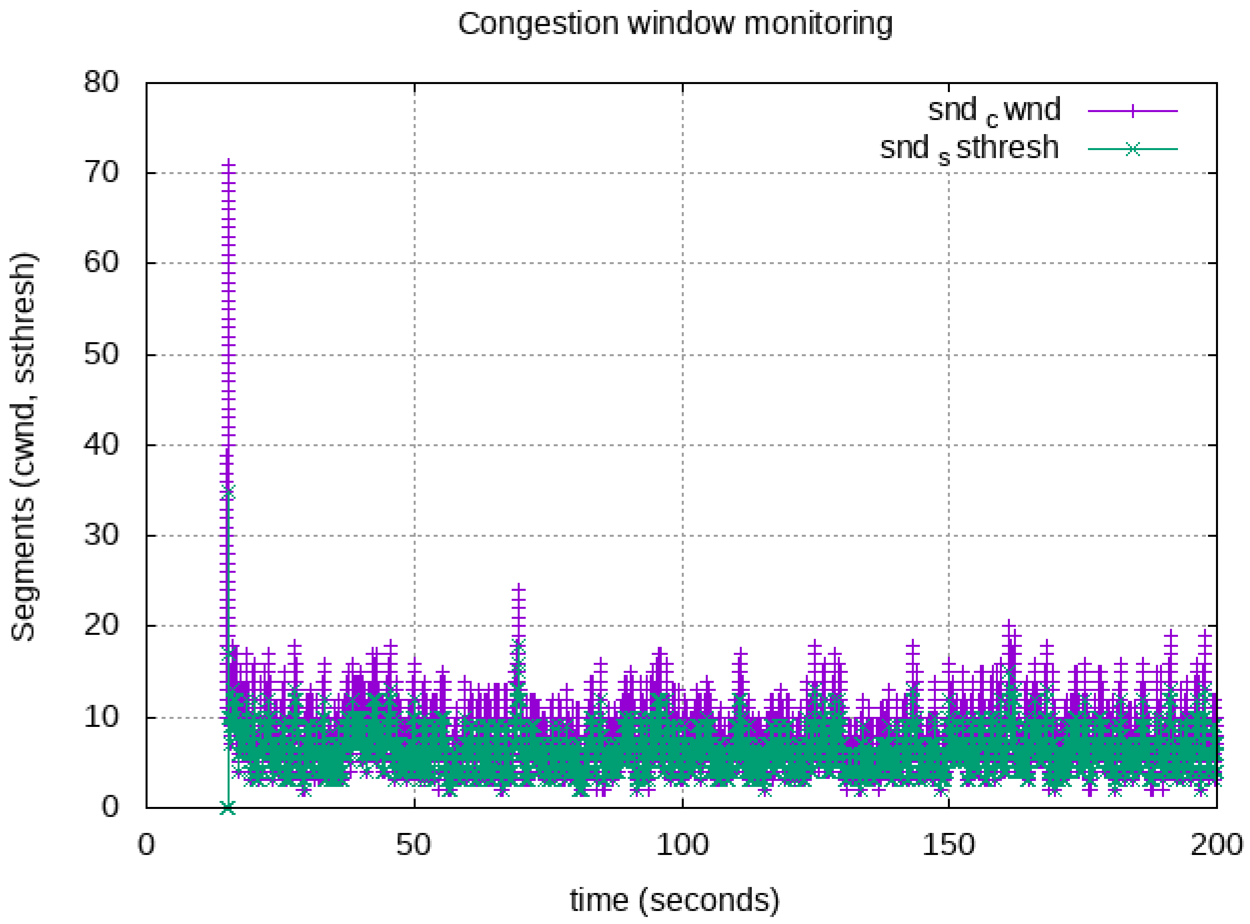


Figure 7: T2Q5 CWND and ssthresh 0 to 200 seconds with packet loss

Q2.6 Compare this graph with the graph of

In this graph, the congestion window never reaches the same levels as in previous exercises. At the peak of slow start, it is just over $70 * MSS$. Here we see a very fast oscillation between fast recovery and congestion avoidance. It does not go back to slow start, because it never starts again with 1MSS. This implies the throughput is much lower than in subsection .

Q2.7 Zoom in the graph of this scenario (plot some parts of this scenario in a short duration, 10 or 20 seconds). Briefly explain the changing process.

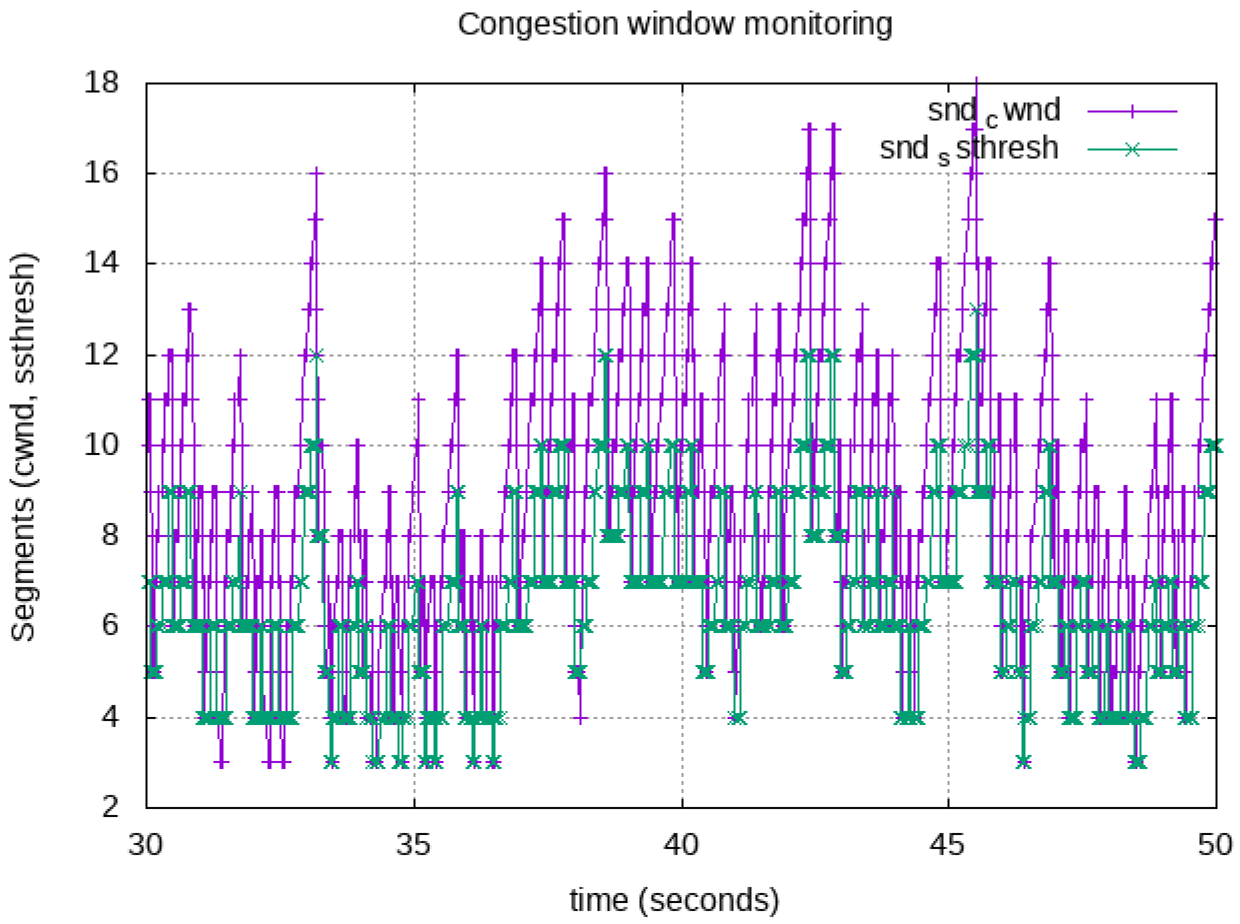
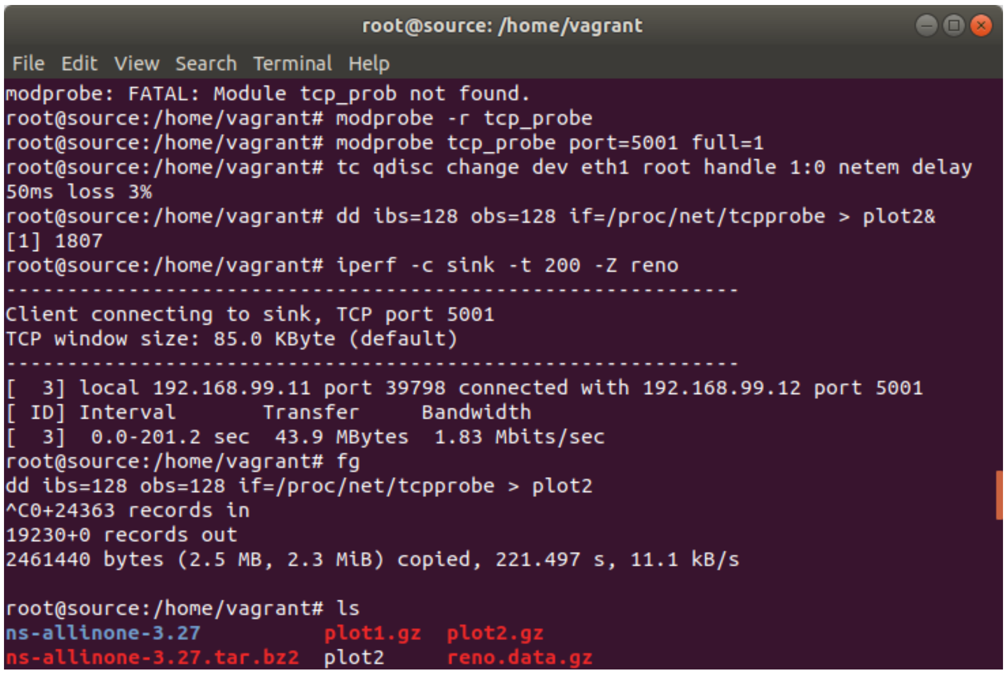


Figure 8: T2Q7 CWND and ssthresh 30 to 50 seconds with packet loss zoomed in

Due to the relatively high percentage of packet loss, this connection is quite unstable. We see the same pattern as previously, oscillating between congestion avoidance and fast recovery, but the lowest points are as much as 8 times lower than the peaks. Previously, the highest difference observed was around 2.8 times.

Q2.8 Show a screen capture of the real throughput in this scenario.


root@source: /home/vagrant
File Edit View Search Terminal Help

```
modprobe: FATAL: Module tcp_prob not found.
root@source:/home/vagrant# modprobe -r tcp_probe
root@source:/home/vagrant# modprobe tcp_probe port=5001 full=1
root@source:/home/vagrant# tc qdisc change dev eth1 root handle 1:0 netem delay
50ms loss 3%
root@source:/home/vagrant# dd ibs=128 obs=128 if=/proc/net/tcpprobe > plot2&
[1] 1807
root@source:/home/vagrant# iperf -c sink -t 200 -Z reno
------------------------------------------------------------
Client connecting to sink, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.99.11 port 39798 connected with 192.168.99.12 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-201.2 sec  43.9 MBytes  1.83 Mbits/sec
root@source:/home/vagrant# fg
dd ibs=128 obs=128 if=/proc/net/tcpprobe > plot2
^C0+24363 records in
19230+0 records out
2461440 bytes (2.5 MB, 2.3 MiB) copied, 221.497 s, 11.1 kB/s

root@source:/home/vagrant# ls
ns-allinone-3.27          plot1.gz  plot2.gz
ns-allinone-3.27.tar.bz2  plot2     reno.data.gz
```

Figure 9: T2Q8 Throughput of Reno algorithm

Q2.9 Plot a graph showing CWND and ssthresh versus time with all the data you get.

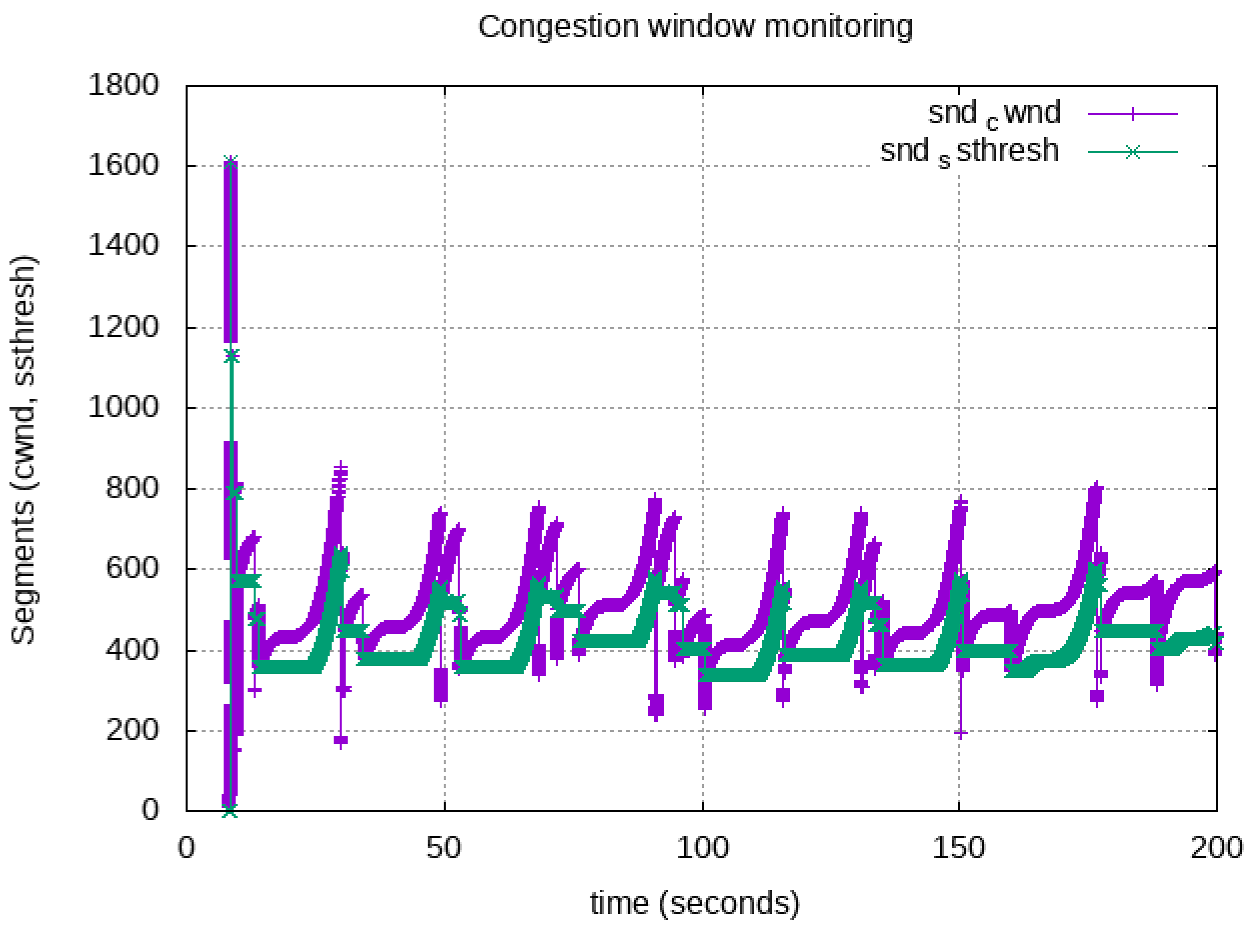


Figure 10: T2Q9 CWND and ssthresh 0 to 200 seconds (CUBIC)

Q2.10 Compare this graph with the graph of

The main difference which catches the eye, is that it has differently-shaped, smooth curves in the congestion window. When looking up how CUBIC works, we see this pattern explained many times as being fast growth upon reduction on a packet loss event, and slower probing for the maximum throughput.

Q2.11 Plot a graph showing CWND and ssthresh versus time with all the data you get.

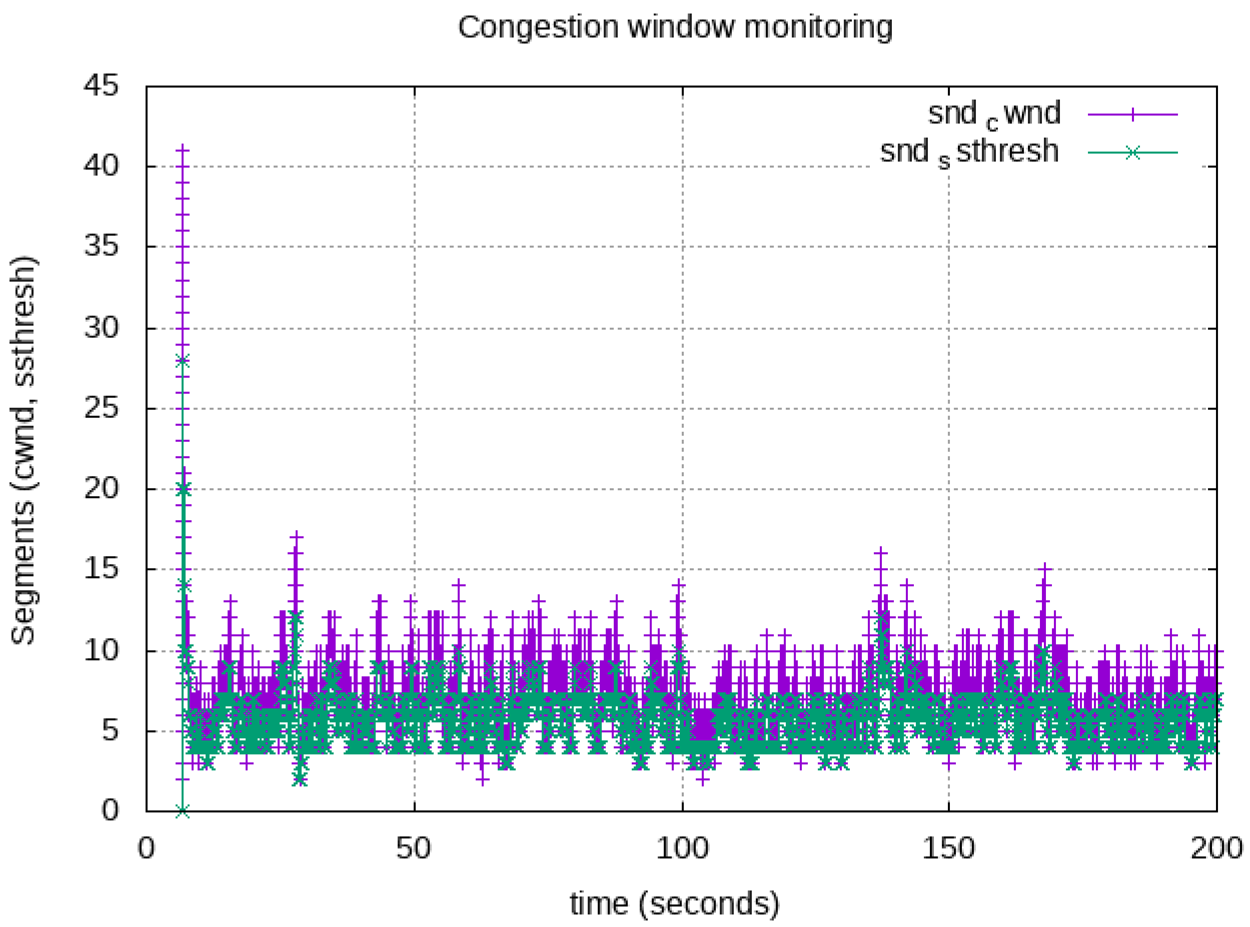


Figure 11: T2Q11 CWND and ssthresh 0 to 200 seconds (CUBIC) with packet loss

Q2.12 Compare this graph with the graph of scenario three and show the differences.

CUBIC generally seems less aggressive. In the slow start state, it does not reach as high as Reno ($42 * MSS$ versus $71 * MSS$). In the later states, it decreases less far and usually goes down to $4 * MSS$ or sometimes $3 * MSS$. Reno shows more fluctuations than CUBIC; therefore, CUBIC seems to be more stable in those circumstances.

Q2.13 Zoom in the graph of this scenario (plot some parts of this scenario in a short duration, 10 or 20 seconds). Briefly explain the changing process and compare it with the graph of

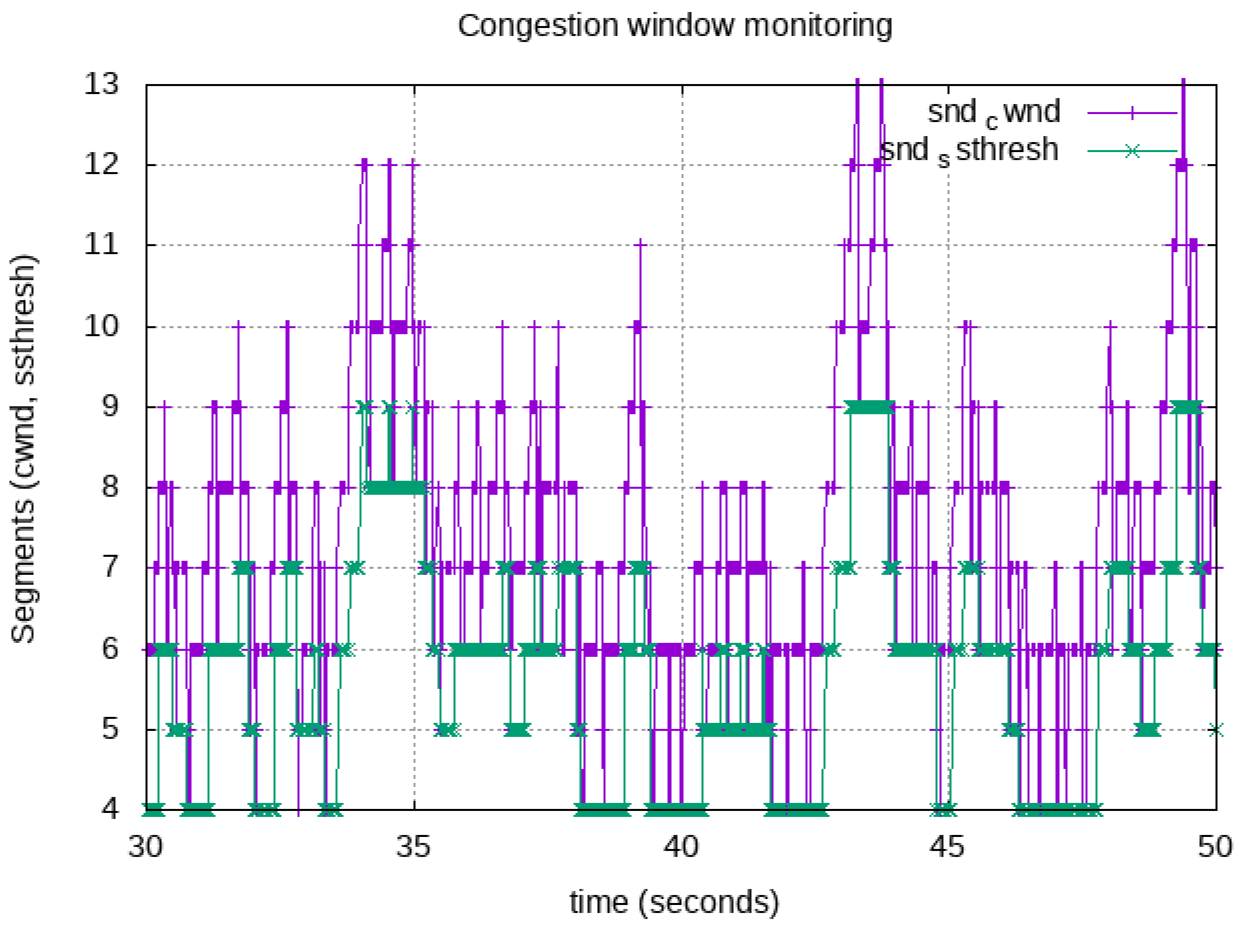
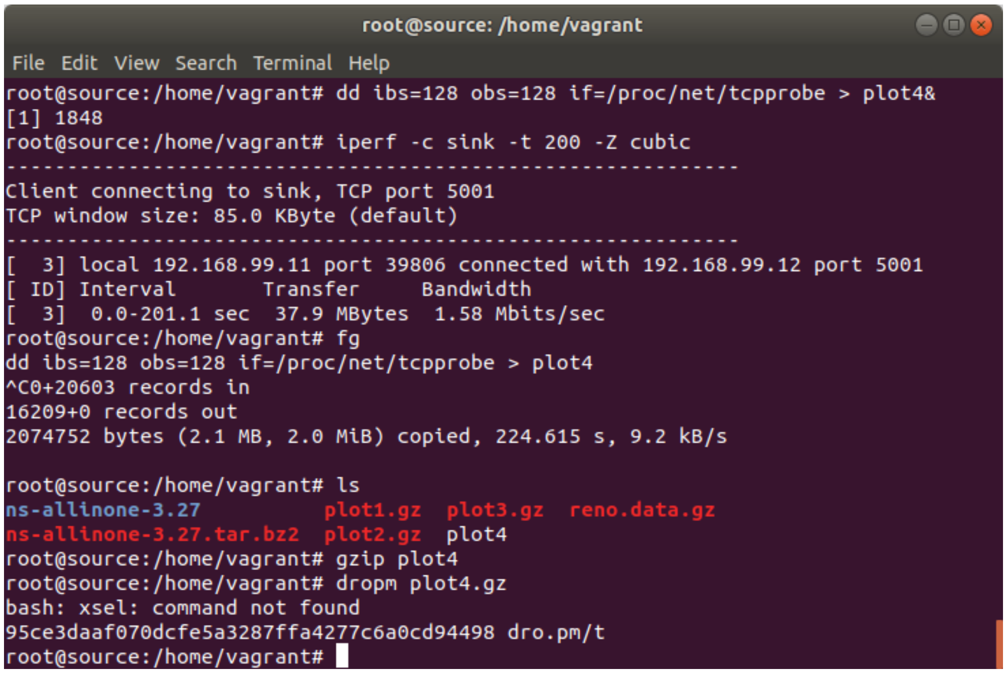


Figure 12: T2Q13 CWND and ssthresh 30 to 50 seconds (CUBIC) with packet loss zoomed in

Here, too, we see that CUBIC is more stable than reno. The difference between the minimum and maximum value, is higher in Reno. The frequency of the state changes is also higher in Reno.

Q2.14 Show a screen capture of the real throughput and compare it with throughput of


root@source: /home/vagrant
File Edit View Search Terminal Help

```
root@source:/home/vagrant# dd ibs=128 obs=128 if=/proc/net/tcpprobe > plot4&
[1] 1848
root@source:/home/vagrant# iperf -c sink -t 200 -Z cubic
------------------------------------------------------------
Client connecting to sink, TCP port 5001
TCP window size: 85.0 KByte (default)
------------------------------------------------------------
[  3] local 192.168.99.11 port 39806 connected with 192.168.99.12 port 5001
[ ID] Interval       Transfer     Bandwidth
[  3]  0.0-201.1 sec  37.9 MBytes  1.58 Mbits/sec
root@source:/home/vagrant# fg
dd ibs=128 obs=128 if=/proc/net/tcpprobe > plot4
^C0+20603 records in
16209+0 records out
2074752 bytes (2.1 MB, 2.0 MiB) copied, 224.615 s, 9.2 kB/s

root@source:/home/vagrant# ls
ns-allinone-3.27          plot1.gz  plot3.gz  reno.data.gz
ns-allinone-3.27.tar.bz2  plot2.gz  plot4
root@source:/home/vagrant# gzip plot4
root@source:/home/vagrant# dropm plot4.gz
bash: xsel: command not found
95ce3daaf070dcfe5a3287ffa4277c6a0cd94498 dro.pm/t
root@source:/home/vagrant# 
```

Figure 13: T2Q14 Throughput of CUBIC algorithm

Reno has a higher throughput than CUBIC.

Q3.1 Explain what an LFN network is. Change the simulation parameters to your likings and demonstrate that TcpNewReno is not suitable for LFN networks.

An LFN ('elephant') network has a DBP (Delay-Bandwidth Product) which is 'significantly' larger than 10^5 bits, i.e. multiplying the bandwidth and the latency is larger than 12 500 bytes per second. It stands for 'long, fat network'.

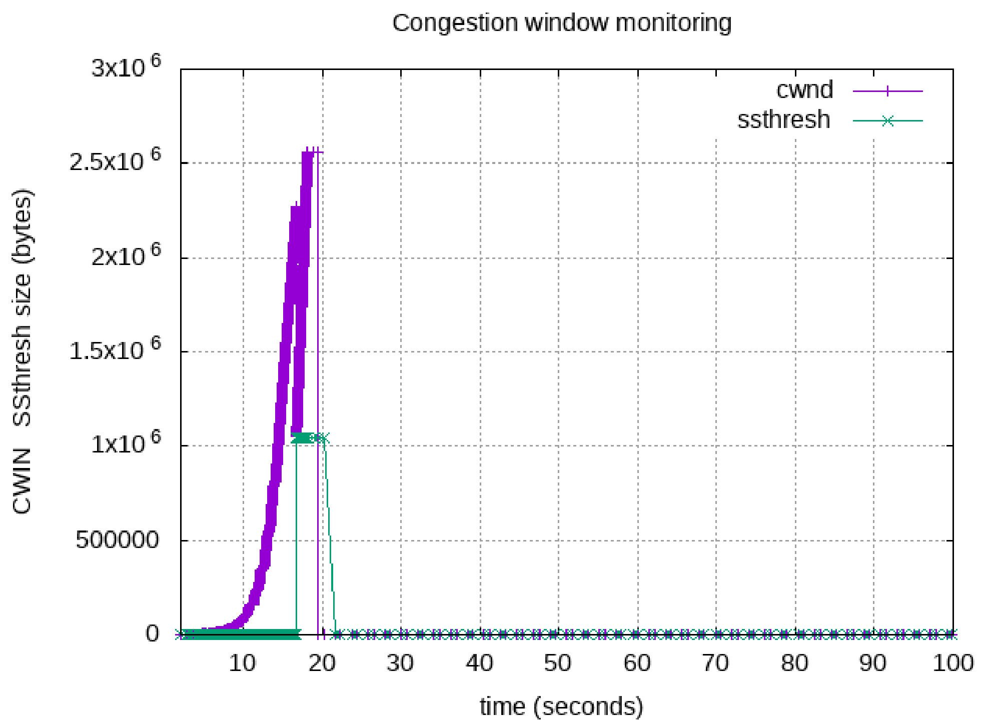


Figure 14: T3Q1 LFN Enabled on TCPNewReno 0 to 100 seconds

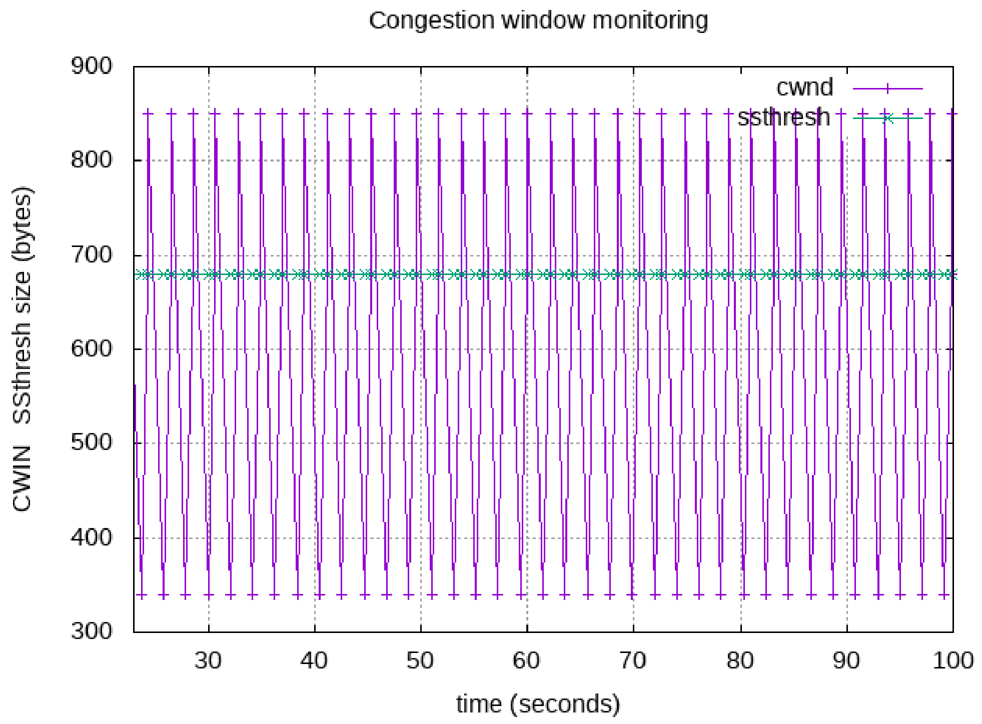


Figure 15: T3Q1 LFN Enabled on TCPNewReno 23 to 100 seconds

Figure 14 shows a long slow start and congestion avoidance phase of approximately 20 seconds, which is followed by a low throughput. This is depicted in figure 15. We configured the network to have a throughput of 500 Mbps and a delay of 300ms. We see that TCPNewReno does not reach 500 Mbps by far. Therefore TCPNewReno is not suitable for LFN networks.

Q3.2 Explain SACK does. Change the simulation parameters to your likings and demonstrate the performance improvement with SACK.

Defined in RFC 2018, SACK allows receivers to indicate they have received segments which are not contiguous.

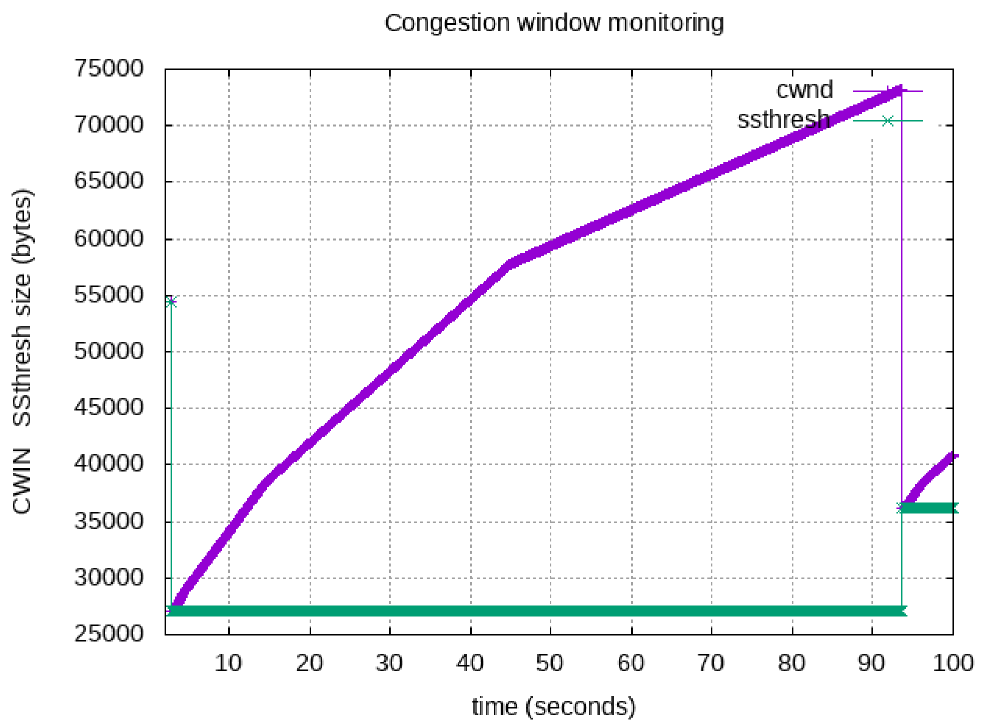


Figure 16: T3Q2 SACK Enabled on TCPNewReno

According to figure 16 higher bandwidths are reached and fewer fluctuations are shown during the simulation, compared to the previous experiments without SACK enabled.

Q3.3 Explain with TCP fairness is. Show the effect of multiple flows in the simulation.

As described by Gorp and Gommans[1]:

> ,,In datacenters, multiple tenants share a finite amount of bandwidth. When an uplink is saturated, packets that overflow available buffer space will be dropped and have to be retransmitted by the sender. When one tenant chooses to retransmit faster than others, they have an advantage as they will make it though more quickly. TCP is a connection-oriented protocol which employs congestion control algorithms. These algorithms aim to divide the available bandwidth equally between network flows.
>
> Some congestion control algorithms are more aggressive than others, and it has been reported that some algorithms cause others to back off excessively. Tenants may, perhaps inadvertently, cause unequal bandwidth distributions during times of congestion."

While no case was found of unfairness, other than Microsoft Windows' CTCP (but that behaviour was so strange that its test was labeled inconclusive), some have much better performance in specific scenarios. In figure 17, CUBIC and BIC are shown to perform similarly in high loss conditions. In figure 18, the same situation is shown but with CUBIC and BBR. In figure 19, CUBIC is compared with BIC with 8ms delay and 0% loss. It is shown that the algorithms perform equally and allocate an amount of bandwidth so equal that on this scale, no difference is visible at all during the majority of the test. Any difference visible, is attributed to random variance.

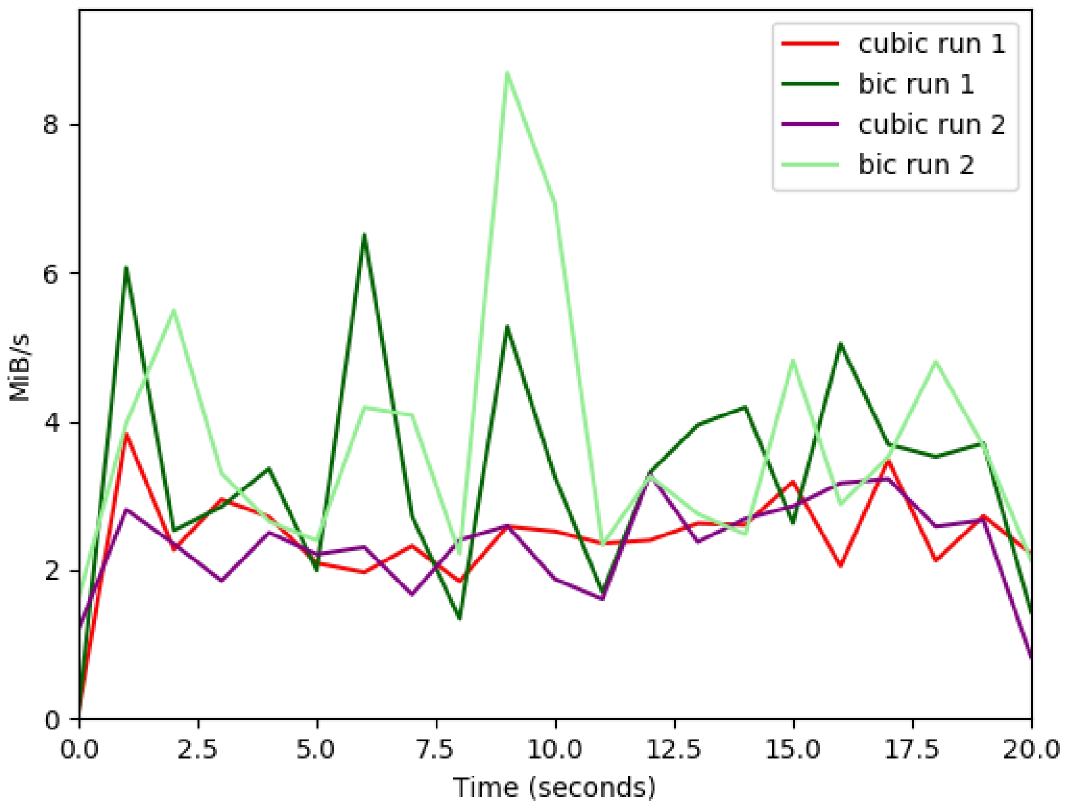


Figure 17: CUBIC vs. BIC, 8ms delay, 1.2% loss

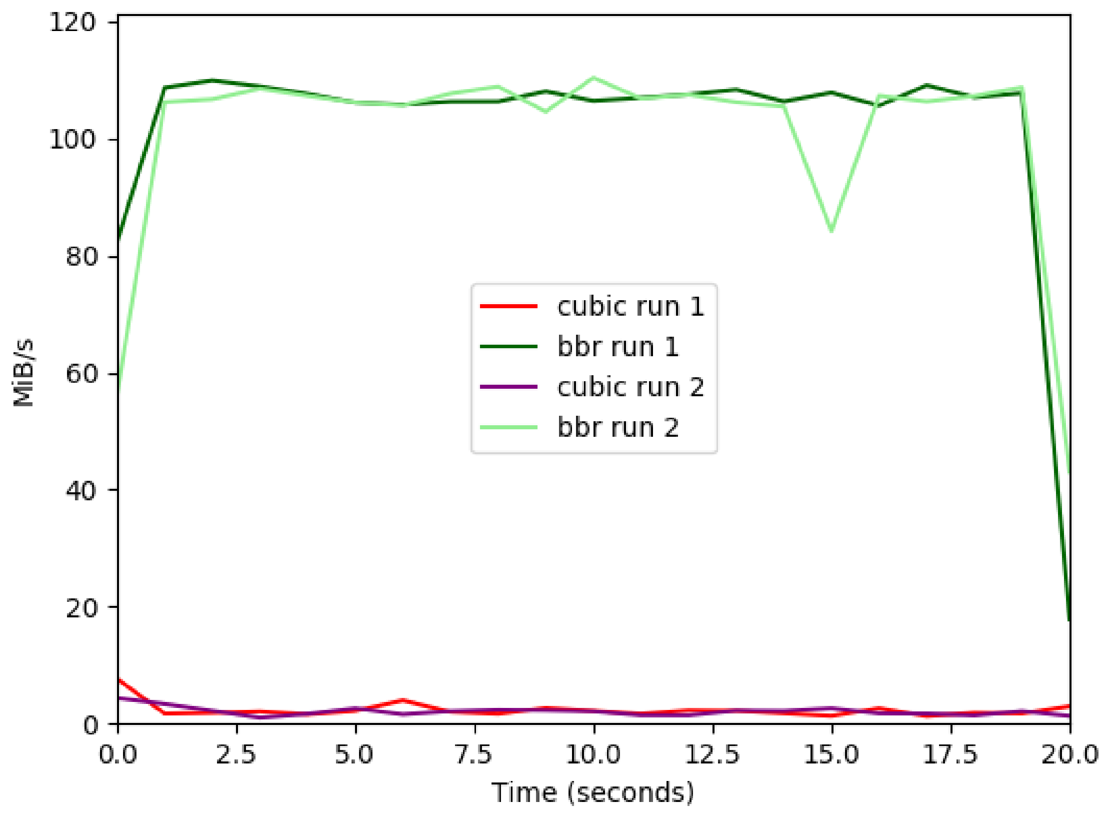


Figure 18: CUBIC vs. BBR, 8ms delay, 1.2% loss

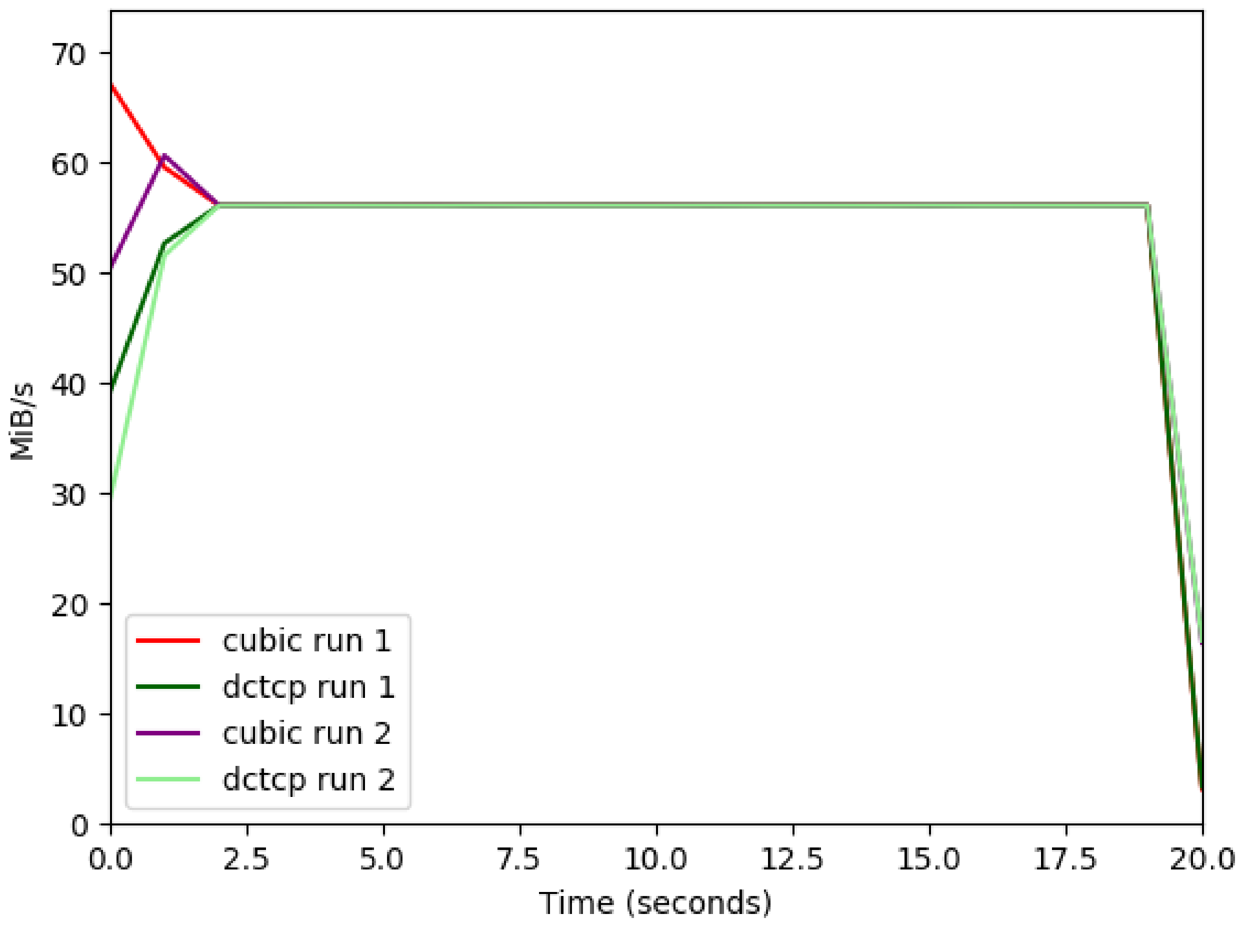


Figure 19: CUBIC vs. BIC, 8ms delay, 0% loss

Q3.4 Replicate scenario 3 of the emulation: packet loss of 3%, delay of 50 ms and transfer duration of 200sec. Use TcpNewReno. Compare the two results.

The plot from `ns3` is shown in figure 20.

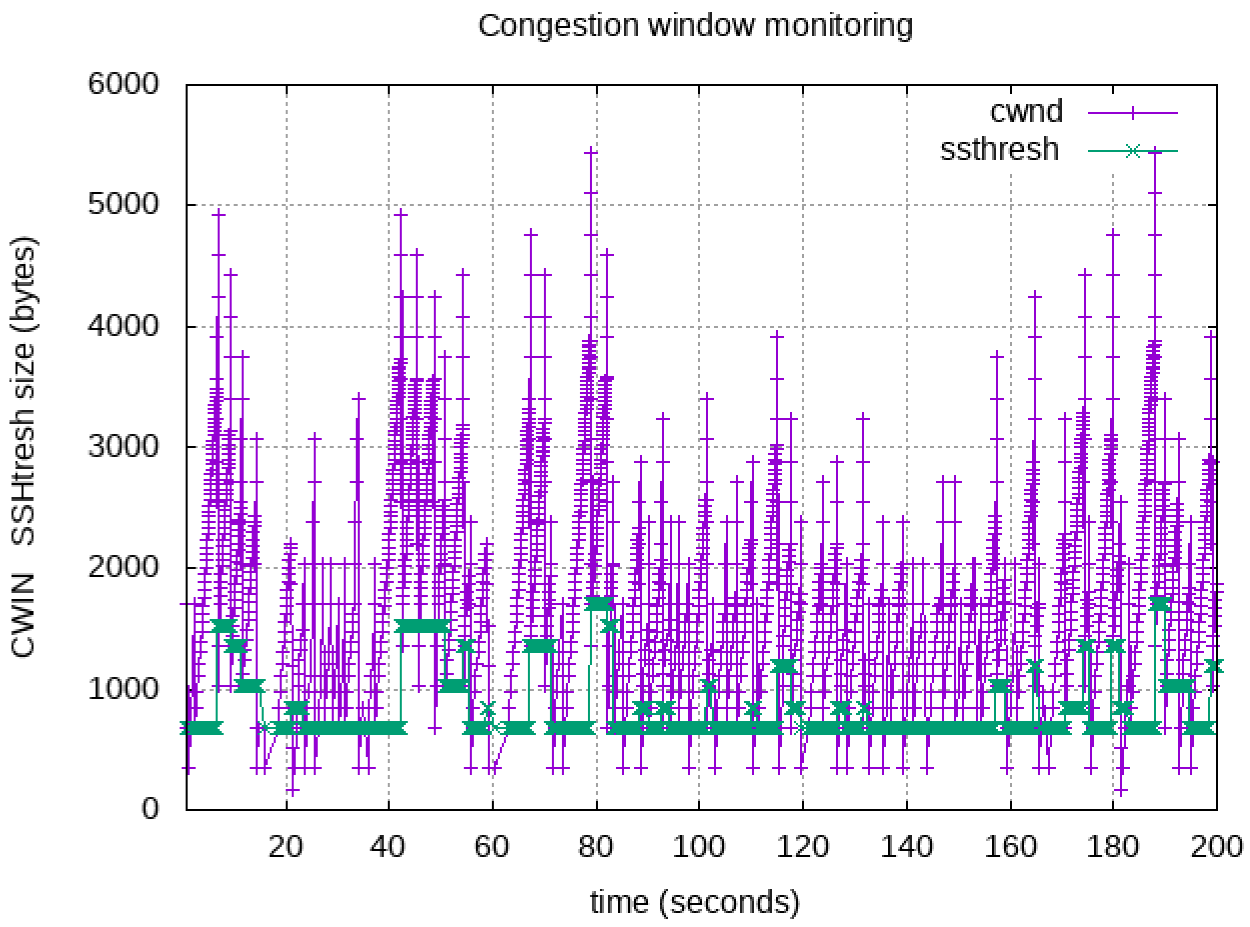


Figure 20: CWND and ssthresh 0 to 200 seconds for TCPNewReno with packet loss and delay

The results in ns3 compared to scenario 3 show different data. Figure 20 shows the size in bytes of $cwnd$ and $ssthresh$, while figure 7 shows the amount of segments. The amount of segments sent according to figure 7 fluctuates more than the size of $cwnd$ in bytes. The default MTU in ns3 is 400 bytes. We see that it subtracts the IP-header, TCP-header and 20 bytes resulting in an MSS of 340 bytes. If we divide the window sizes at multiple points in figure 20 by 340 bytes, the resulting amount of segments corresponds to the amount of segments from figure 7. This is however, not exactly checked due to the density of figure 7.

Q3.5 After these experiments, please briefly describe the difference between simulation and emulation?

> ,,Emulation is the process of mimicking the outwardly observable behavior to match an existing target. The internal state of the emulation mechanism does not have to accurately reflect the internal state of the target which it is emulating.
>
> Simulation, on the other hand, involves modeling the underlying state of the target. The end result of a good simulation is that the simulation model will emulate the target which it is simulating."

From Toybuilder on StackOverflow[2].

In our case, the emulator is NS3 and the simulation is done using iperf.

References

[1] Rick van Gorp and Luc Gommans. *TCP Congestion Algorithms In Datacenters*. URL: `https://github.com/lgommans/lsproj/blob/master/paper/paper-v1.0.1-2017-12-24.pdf`.

[2] Toybuilder. *Simulator or Emulator? What is the difference?* URL: `https://stackoverflow.com/a/1584701/1201863`.