Mini project NetPerf 2024

# Submit to srp@es.aau.dk with subject [NetPerf24][MiniProject][Emil L. H Jens-Ulrik L-M]

# Task 1 - Source coding

# Task 2 - Buffer bloat lab exercise

Description:

Complete the Mininet bufferbloat exercise available at the link  
[https://github.com/mininet/mininet/wiki/Bufferbloat](https://github.com/mininet/mininet/wiki/Bufferbloat%20) and include the results in your report.

Lots of problems with VM, pip, matplotlib, tcpprobe, linux kernel … however we think it is working now.

## Part 2: Web page download - Sketch the TCP CWND

The first objective in the exercise is to measure how long it takes to download a web page from H1 and get a transfer time of 1 second.

The behavior of the CWND at H1 would look like since it is following TCP reno policy, see sketch.

Et billede, der indeholder tekst, diagram, linje/række, Font/skrifttype

Automatisk genereret beskrivelse

Reno features slow start (SS) which uses exponential growth to grow until SSTHRES (8 in plot) is reached then uses additive increase and multiplicative decrease until the transfer is complete. In this case the buffer can almost hold the entire webpage, meaning once the slow start is over, H1 never has to hold back sending. In this short traffic the TCP pretty much stays in the SS-phase.

## Part 3: “Streaming video” - Sketch the TCP CWND and Buffer Occupancy.

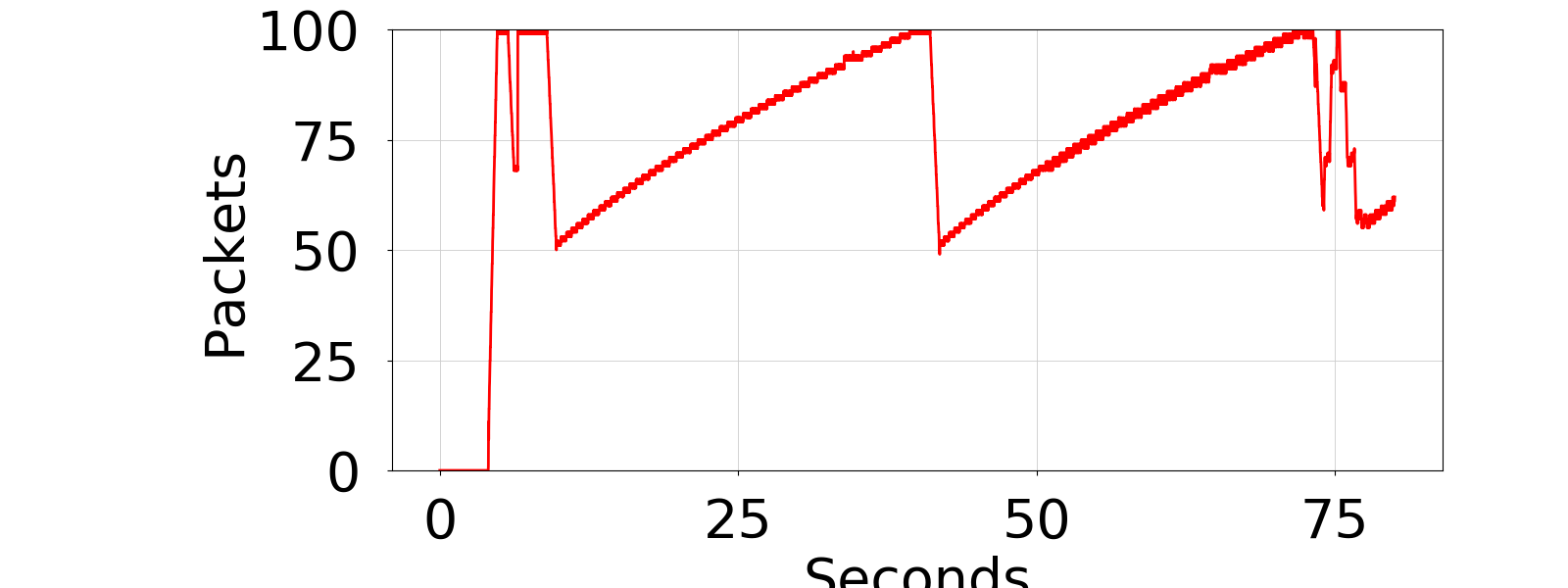
In this part we now set up some traffic between H1 and H2 using iperf. This added traffic will cause congestion on the line and thus the same transfer cannot be achieved anymore. Using the ping command, we can see that the ping has increased and that it goes between 400ms and 800ms in a cycle. This results from the TCP policy and materializes as the TCP sawtooth pattern, see sketch.

Et billede, der indeholder tekst, diagram, linje/række, Font/skrifttype

Automatisk genereret beskrivelseWhen rerunning the webpage download it now takes 7.2 seconds instead of the original 1 second. This is because the buffer in the router of the network gets congested by all of the additional traffic caused by the iperf command causing H1 to halt/slow down the transmission of data to H2 in order for the packet to arrive. Thus the TCP will enter congestion avoidance and go into additive increase mode. In this mode the cwnd will gros slowly and if dup ACK’s is received the cwnd will be set to cwnd/2 and from where the fast recovery will set in. This dance will continue until the transfer is complete.

## Part 4: Measuring the real cwnd and buffer occupancy values.

In this part we do actual measurements of the cwnd and buffer during the iperf traffic and the webpage download. We use restart mininet, setup the iperf traffic and wait 70 seconds (says so in exercise) and do the webpage download. While this happens a monitor script measures the cwnd and buffer, the result is shown in the three figures below. We see initially just the iperf running, then the iperf and wget running and finally the queue throughout the experiment.

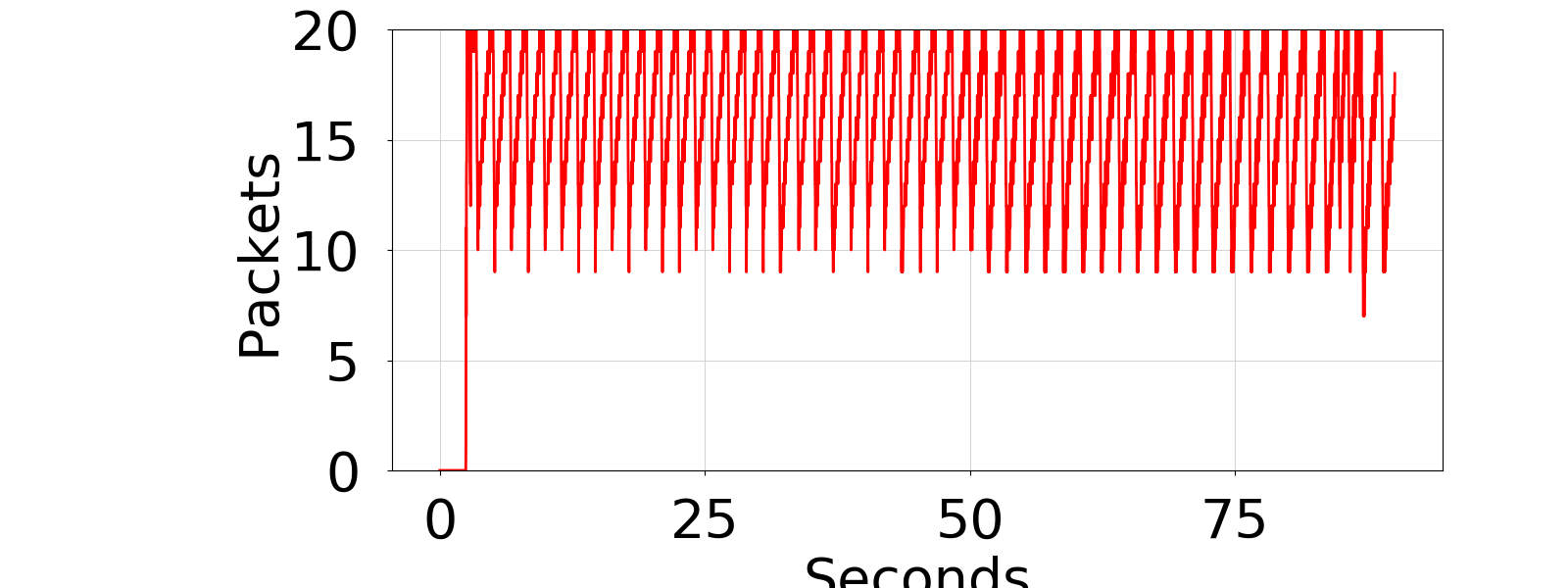
Here we see that there is an initial spike in packets, due to slow start quickly ramping to a high transfer rate. After that we see congestion avoidance kick in and the queue growth is controlled. When the sudden spike happens, it’s caused by the wget traffic trying to get through the buffer.Et billede, der indeholder linje/række, Kurve, diagram, skibakke

Automatisk genereret beskrivelse

This causes the congestion avoidance dance of TCP seen earlier, the wget is not as noticeable here however a small increase in slope of the cwnd can be seen at the end.

## Part 5: Make the router buffer smaller. Reduce it from 100 packets to 20 packets.

We repeat the experiment with downloading the webpage, this time with a smaller buffer and both with and without iperf running. This time running the wget produces two times, 1.2 seconds without iperf and 2.8 seconds with iperf. This is a slight decrease without and a large increase with iperf running. The reason being there are less ”junk” packets blocking the way of the webpage packets, leading to an overall reduction in transfer time when iperf is running. The experiment was logged and the graphs can be seen below:



Here we again see that the buffer quickly fills up and then starts the TCP dance due to congestion avoidacne. At the end where the queue gets a bit more hectiv is when the wget is called and the webpage downloaded.

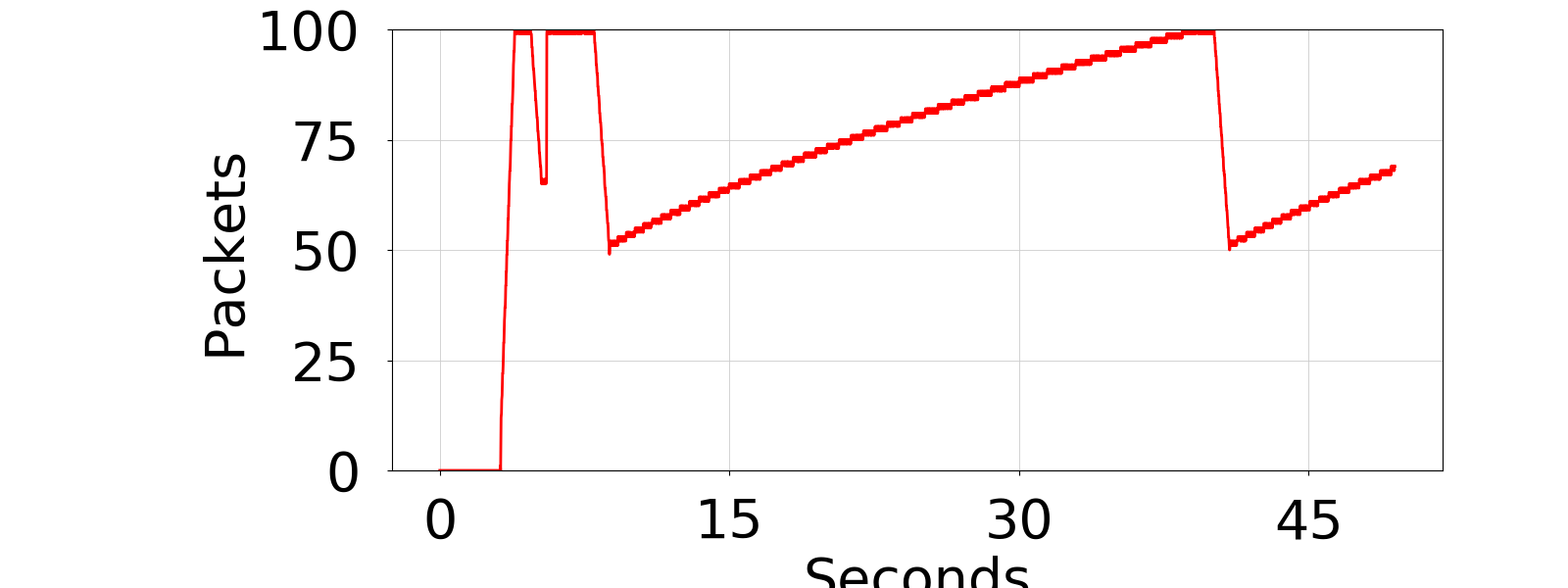
Et billede, der indeholder tekst, linje/række, Kurve, Font/skrifttype

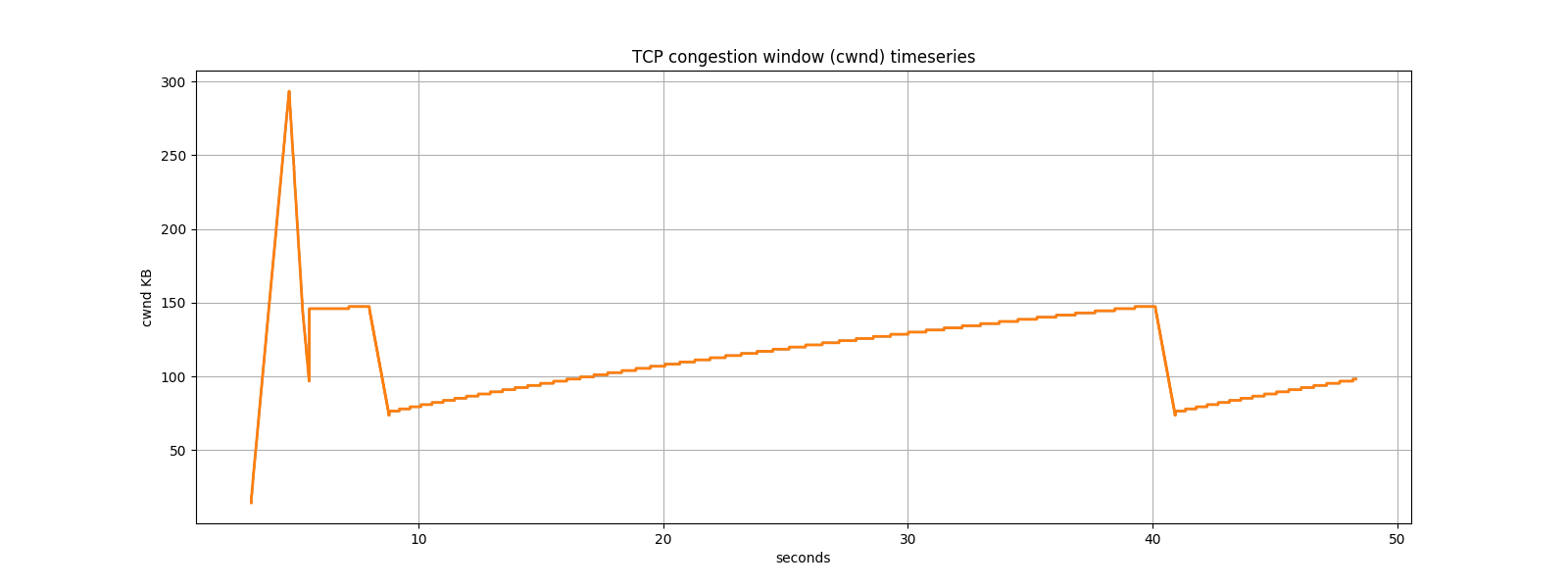
Automatisk genereret beskrivelse

Here we again see the slow start followed by congestion avoidance. We here see a smaller swnd since the buffer is smaller and here it’s really difficult to see the effect of wget since the duration is much shorter due to the decreased queue length.

## Part 6: Different queues

In this final part we split the iperf and wget traffic into two separate queue’s. The split will allow the slow/long lasting traffic from blocking the quick/short flow of downloading the webpage. Thus the ping and download speed will be much less affected by the iperf. Due to this we get a wget time of 1.1 seconds without iperf and a wget time of 1.2 seconds with iperf running. This experiment was also logged:



In this experiment the queue is large again, however due to the split this is only the iperf queue.

And this performs similarly to earlier however with the benefit that the quick/short traffic gets a way better opportunity to pass unhindered.

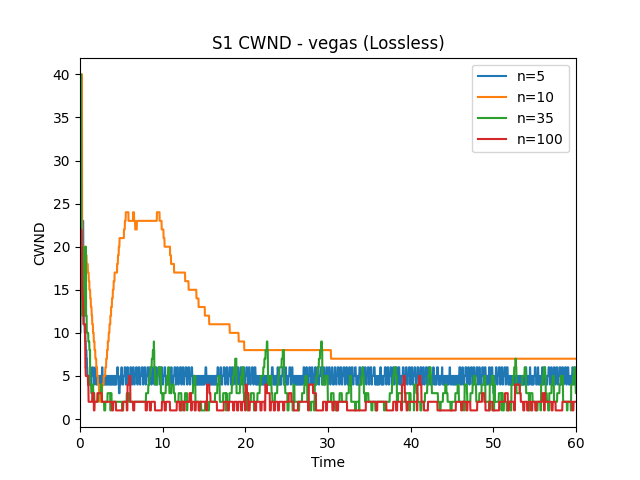
# Task 3 - Dumbbell and types of TCP (lossless)

In this task we are to construct the shown dumbbell network structure and simulate the CWND with different configurations, loss and TCP congestion control algorithms. For this purpose, Mininet will again be used, we repurpose the code in the bufferbloat exercise to now construct the dumbbell structure. With this we can use the cwnd monitor from the bufferbloat exercise to measure the performance for different source/destination counts and with/without loss. In this task we test TCP reno/vegas/cubic for the counts: 5, 10, 35, 100. The results can be seen below:

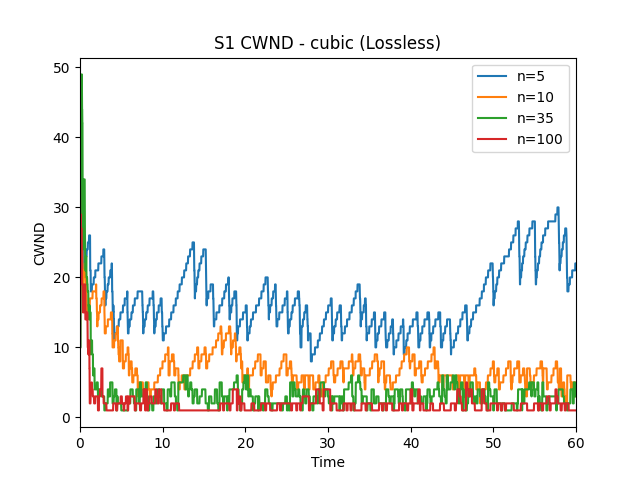
Et billede, der indeholder tekst, skærmbillede, Kurve, diagram

Automatisk genereret beskrivelse

In the case of TCP Reno, we clearly see the initial slow start scale exponentially followed by the cycle of AI and MD leading stairway structure. We see that the CWND is lowered for more users of the network since they must share the bottleneck transfer rate between the routers.



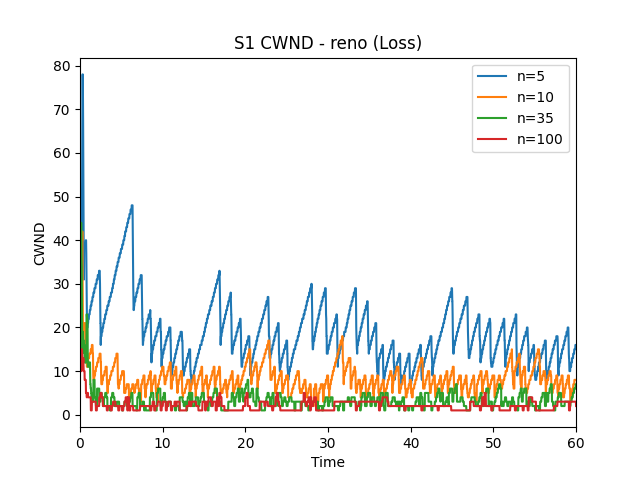
Here we see TCP Vegas which does not change between AI and MD, but uses the measured RTT to adjust the CWND. In this way TCP Vegas tries to keep the CWND constant. This can be seen by the lack of stair structure like in Reno.



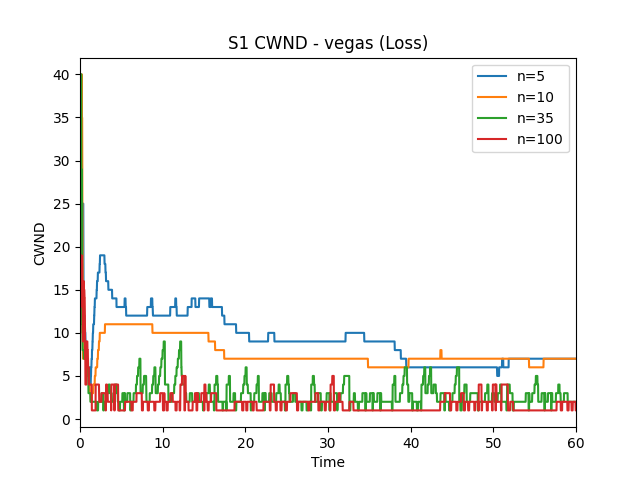
Finally, we have TCP Cubic, which tries to be more aggressive compared to Reno. The cubic growth is faster than the AI when increasing the CWND. In Cubic the growth slows as the previous CWND limit is reached, thus spending more time near maximum throughput. Cubic still uses MD, and thus the result becomes a Reno like structure, but Cubic the CWND is generally larger equaling more throughput.

# Task 4 - Dumbbell and types of TCP (loss)

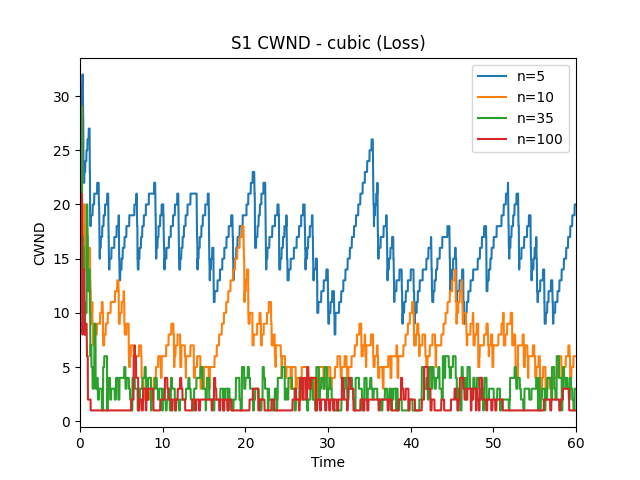
In this task we are to repeat the simulations now introducing a 1% packet loss rate, the new results can be seen below:



In the case of loss, it can be seen the CWND is lowered a bit since more MD occurs lowering the CWND more times during the 60 sec simulation.



Here it can be seen that the packet losses cause the RTT to suffer leading the TCP Vegas to lower the CWND.



Here it can be seen that the effect of the packet loss is less since the aggressive nature of Cubic recovers more quickly then Reno and Vegas.

# Task 5 (Bonus) - BBR simulation (loss/lossless)

In this task we try to use the more modern TCP congestion control algorithm BBR (by Google). This however produces an error:

Setting TCP congestion control to bbr

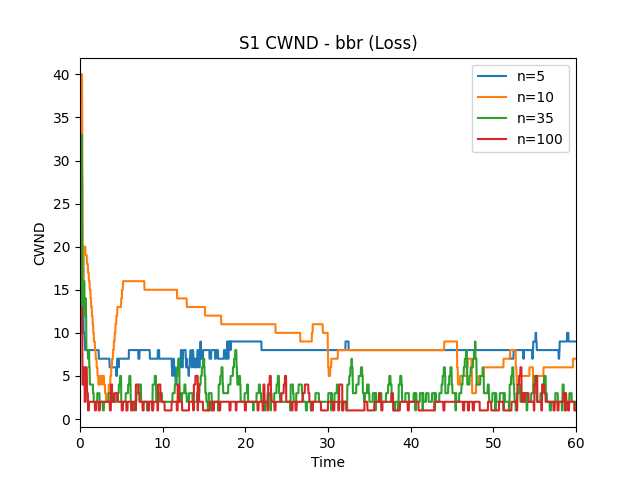
sysctl: setting key "net.ipv4.tcp\_congestion\_control": No such file or directory

net.ipv4.tcp\_congestion\_control = bbr

However, the code still executes and produces a result:

Et billede, der indeholder tekst, skærmbillede, diagram, linje/række

Automatisk genereret beskrivelse



The performance looks quite like Vegas. BBR tries to keep the latency low, thus in the low source/destination count cases the CWND is smaller than compared with TCP Reno and Cubic which try to maximize throughput and not latency.