Performance Comparison of TCP Versions

Authors: Harsh Gaurang Kapadia (harshk@bu.edu), Yuxuan Huang (yxuhuang@bu.edu), Pranesh Jayasundar (praneshj@bu.edu), Haniel Edward Jacob Thomson (hanielj@bu.edu)
GitHub Repository: HarshKapadia2/tcp-version-performance-comparison
Demo Video: https://www.youtube.com/watch?v=s_6OOjMOxpQ
GENI Slice Name: `harsh-kapadia-tcp`

1. Problem Statement

1.1. Definition

The Transmission Control Protocol (TCP) is one of the most widely used protocols on the internet for reliable data transfer. Over the years, there has been a lot of research into improving the performance of TCP when it comes to utilizing a maximum of the available capacity (bandwidth) of the connection and at the same time keeping the newly developed versions fair with the previous versions in terms of the available throughput for different TCP versions sharing the same connection.

The TCP versions explored
1. TCP Reno
2. TCP CUBIC
3. TCP Vegas
4. TCP BBR

1.2. Learning Outcomes

1.2.1. Learning and comparing how different TCP versions respond to network congestion.
1.2.2. Comparing how fair different TCP versions are to each other.

1.3. Experiments

Two experiments were conducted to meet the learning outcomes mentioned in section 1.2 above.

1.3.1. Experiment No. 1

Plot the Time vs Congestion Window (CWND) graphs for each TCP version.
This fulfills learning outcome 1.2.1.

1.3.2. Experiment No. 2

Plot the Time vs Congestion Window (CWND) and Time vs Throughput (TPUT) graphs for two TCP versions running simultaneously.
This fulfills learning outcome 1.2.2.

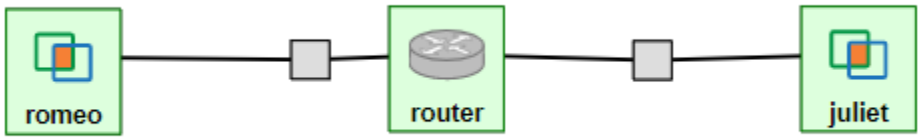
2. Design and Execution

The TCP Congestion Control blog was used to obtain the infrastructure schema, basic scripts and methodology to conduct the experiments.

TCP Congestion Control blog: witestlab.poly.edu/blog/tcp-congestion-control-basics

2.1. Setup Diagram

RSpec file:
witestlab.poly.edu/blog/tcp-congestion-control-basics/#runmyexperiment



For all the experiments below, **Romeo is the Sender** and **Juliet is the Receiver**.

2.2. Experimental Methodology

The Router's Buffer Capacity (Rate) and Buffer Size need to be set to 1 Mbps and 0.1 MB respectively to get comparable results. The package `iperf3` also needs to be installed to simulate network traffic. All the commands for this can be found in the 'Set up experiment' section of the blog.

Set up experiment:
witestlab.poly.edu/blog/tcp-congestion-control-basics/#setupexperiment

2.2.1. Experiment No. 1

2.2.1.1. Method

1. Network traffic is sent from Romeo to Juliet through the Router.
2. Three parallel flows of data are sent from the sender over

60 seconds using `iperf3`. This is done for each considered version of TCP individually.

3. The Throughput (TPUT) and Congestion Window (CWND) values are recorded for each algorithm using `iperf3` and `ss`.
4. Using the Shell and R scripts provided by the blog, a graph for Time vs CWND is plotted for the three parallel flows.

2.2.1.2. Scripts and Commands

1. Shell script: witestlab.poly.edu/blog/tcp-congestion-control-basics/#generatingdata
2. R script: witestlab.poly.edu/blog/tcp-congestion-control-basics/#visualization
3. Commands to execute: witestlab.poly.edu/blog/tcp-congestion-control-basics/#generatingdata

2.2.2. Experiment No. 2

2.2.2.1. Method

1. Two simultaneous flows of different TCP versions are sent from the sender for 60 seconds using `iperf3`.
2. This simultaneous flow is executed for permutations of the four TCP versions mentioned in section 1.1.
3. After modifying the Shell and R scripts provided by the blog, graphs for Time vs TPUT and Time vs CWND were plotted for the simultaneous flows.

2.2.2.2. Scripts and Commands

The Shell and R scripts for this experiment had to be modified from the original scripts provided by the blog.

1. Modified scripts: github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/scripts
2. Commands to execute: https://witestlab.poly.edu/blog/tcp-congestion-control-basics/#additionalexerciseslowdelaycongestioncontrol (2nd half of the section in the link)

3. Results

3.1. TCP Reno (Experiment No. 1)

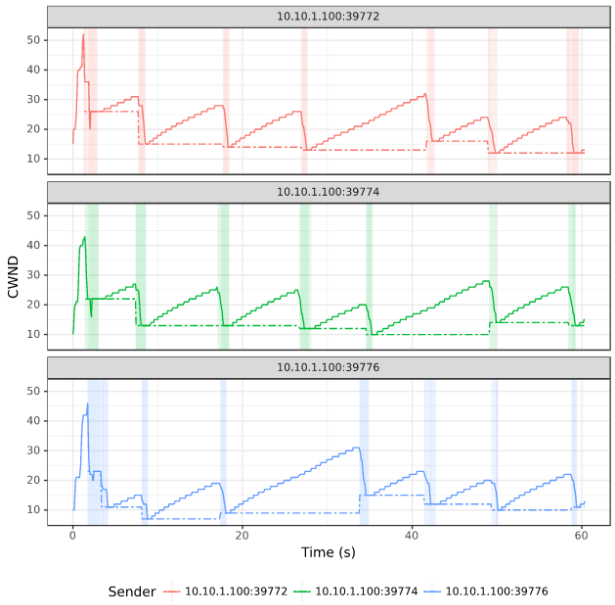
3.1.1. Hypothesis

TCP Reno goes through the Slow Start, Congestion Avoidance (AIMD) and Fast Recovery phases during Congestion Control.

The Time vs CWND graph is expected to start with the exponential Slow Start phase, which is used to find the point of congestion (Slow Start Threshold - SSTHRESH) as quickly as possible, to be able to utilize the link to its maximum capacity. After this, the Slow Start phase will be seen only if any packets time out.

If we only receive three duplicate acknowledgements, we expect to see Congestion avoidance and Fast Recovery stages repeating every time we have such a condition.

3.1.2. Result and Analysis



As expected, we could initially see an exponential growth of the CWND, which is the Slow Start phase. In our measurements, we saw the Slow Start phase only at the start of the flows, but not after that indicating that there were no packets that timed out.

Packet retransmissions are indicated by the coloured vertical lines in the

graphs and we can see post the Slow Start phase that a Sawtooth pattern is visible. This is the Fast Recovery phase repeating every time three duplicate acknowledgements are received.

The dotted line indicates the SSTHRESH values for every phase.

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/reno

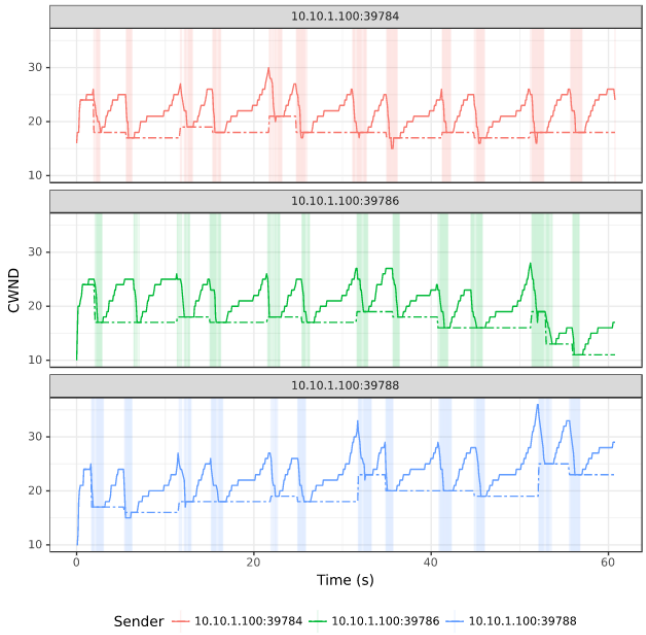
3.2. TCP CUBIC (Experiment No. 1)

3.2.1. Hypothesis

TCP CUBIC uses a cubic function to regulate CWND, which aggressively increases the CWND in a convex fashion and once the Slow Start Threshold (SSTHRESH) is passed, it increases aggressively in a concave fashion.

In comparison to TCP Reno, it is expected that TCP CUBIC will have a more aggressive increase of the CWND, but will result in more retransmissions due to reaching SSTHRESH more frequently.

3.2.2. Result and Analysis



As expected, the CWND is cubic and much more aggressive than TCP

Reno.

From the graphs of TCP Reno and TCP CUBIC, it can be seen that the loss events for TCP CUBIC are more frequent than for TCP Reno, as expected. So, TCP CUBIC is able to reach optimum utilization much faster and more frequently than TCP Reno.

Also, as expected, the outputs of `iperf3` reveal that TCP CUBIC had 119 retransmissions, while TCP Reno had 97.

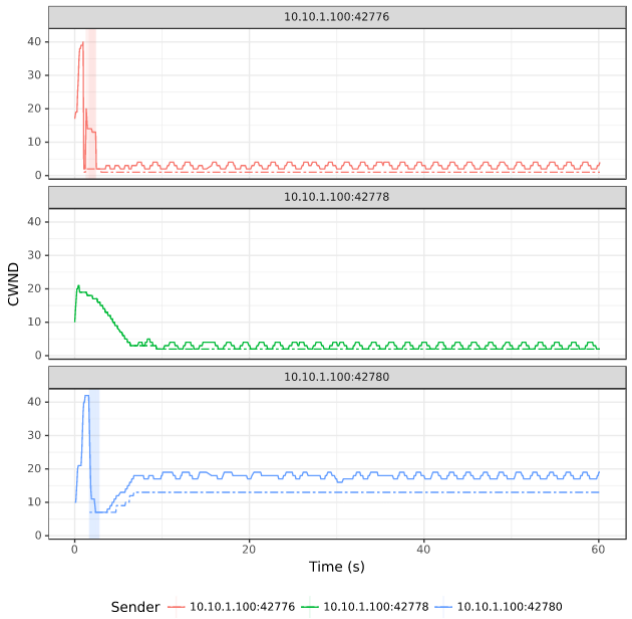
More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/cubic

3.3. TCP Vegas (Experiment No. 1)

3.3.1. Hypothesis

TCP Vegas is a Delay-based Algorithm which modifies the CWND size based on the Round-Trip Time (RTT) values calculated on-the-fly and keeps it steady between a certain range, unlike TCP Reno and TCP CUBIC, which are Loss-based Algorithms. This implies that TCP Vegas should not suffer from retransmissions.

3.3.2. Result and Analysis



Although there were a few retransmissions (29 as per `iperf3`), they are far fewer than TCP Reno (97 retransmissions) or TCP CUBIC (119 retransmissions).

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/vegas/itr-2

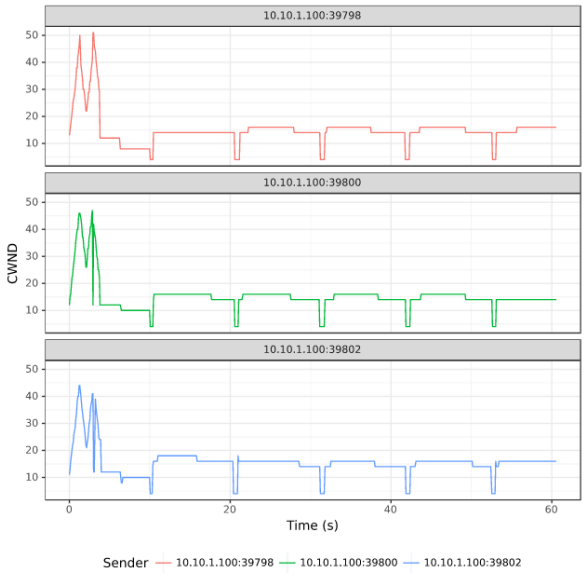
3.4. TCP BBR (Experiment No. 1)

3.4.1. Hypothesis

TCP BBR (Bottleneck Bandwidth and Round-Trip Propagation Time) is a Delay-based and Model-based Algorithm. TCP BBR uses measurement for the network's Bottleneck Bandwidth (BB) and Round Trip Propagation Time to build a model which helps determine the data sending (pacing) rate.

Expected TCP BBR phases in the graph:
1. Startup Stage (Sharp increase in CWND to probe BB by causing queuing at Bottleneck connection)
2. Drain Stage (Inverse of Startup Stage to remove queuing at the Bottleneck connection)
3. Probe Bandwidth (ProbeBW) Stage (Steady state which monitors RTT)

3.4.2. Result and Analysis



As expected, the sharp rise in the CWND at the start is the Startup Stage, the dip in CWND after that is the Drain Stage and the steady state after that is the ProbeBW Stage.

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/bbr

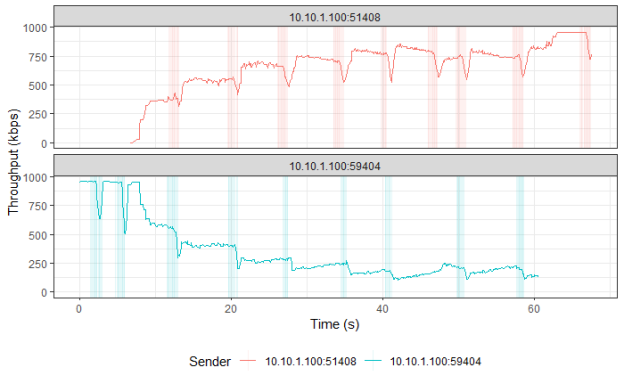
3.5. TCP Reno vs TCP CUBIC (Experiment No. 2)

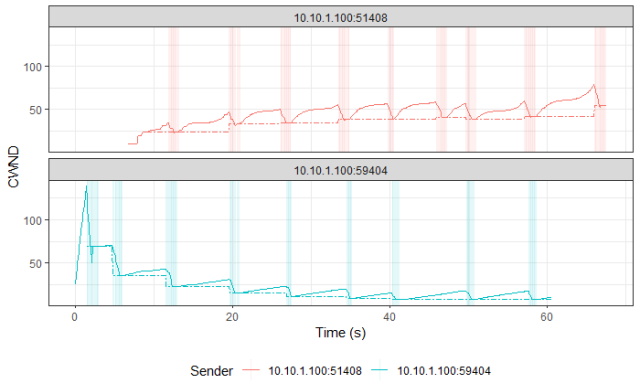
3.5.1. Hypothesis

As TCP CUBIC is more aggressive than TCP Reno, it is expected that it will not be fair to TCP Reno and will dominate the connection's TPUT.

3.5.2. Result and Analysis

NOTE: TCP CUBIC followed by TCP Reno in both graphs.





As expected, TCP CUBIC bullies TCP Reno and dominates the connection, thus behaving unfairly with TCP Reno.

As TCP CUBIC increases its CWND very aggressively, it reaches the

point of congestion faster and in the process transfers more data than TCP Reno. The faster CWND increase by TCP CUBIC keeps happening and over time consumes the available buffer capacity at the bottleneck queue, which forces TCP Reno to keep reducing its CWND, which implies lesser TPUT for TCP Reno with time.

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/reno-vs-cubic

3.6. TCP Reno vs TCP Vegas (Experiment No. 2)

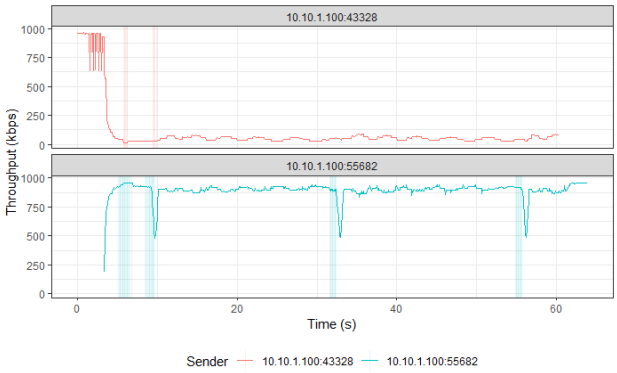
3.6.1. Hypothesis

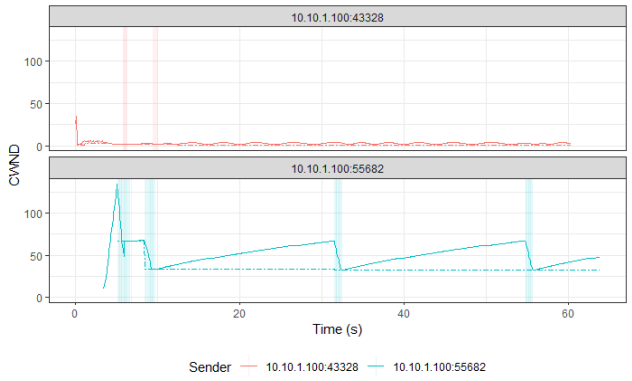
TCP Reno is a Loss-based Algorithm, while TCP Vegas is a Delay-based Algorithm. This implies that TCP Vegas will adjust its CWND based on the RTT that it measures on-the-fly, so that it doesn't lose packets. In contrast, TCP Reno increases its CWND till it loses packets.

These behaviors made us think that TCP Vegas will reduce its CWND as soon as it detects an increase in RTT for its packets that will be cause by the aggressive TCP Reno filling up the Bottleneck queue with its packets.

3.6.2. Result and Analysis

NOTE: TCP Vegas followed by TCP Reno in both graphs.





As expected, TCP Vegas' average TPUT is ~121 kbps (as reported by `iperf3`) and TCP Reno has an average TPUT of ~957 Kbps (as reported by `iperf3`).

TCP Reno is thus not fair to TCP Vegas by any means. TCP Vegas' algorithm is too civil to compete with TCP Reno's algorithm.

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/reno-vs-vegas

Also, **similar results are expected for TCP CUBIC vs TCP Vegas**, as TCP CUBIC is even more aggressive than TCP Reno (which was proved in section 3.5).

3.7. TCP Reno vs TCP BBR (Experiment No. 2)

3.7.1. Hypothesis

TCP BBR actively avoids network congestion (by operating at the optimal 'knee' as seen in the figure below), whereas a Loss-based Algorithm such as TCP Reno waits for a congestion (packet loss) to occur to react to the congestion (by operating at the 'cliff' on the extreme right of the figure below).

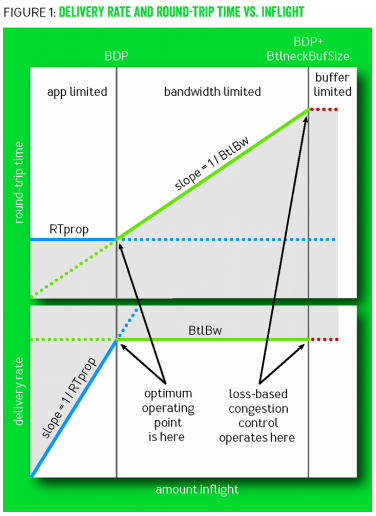
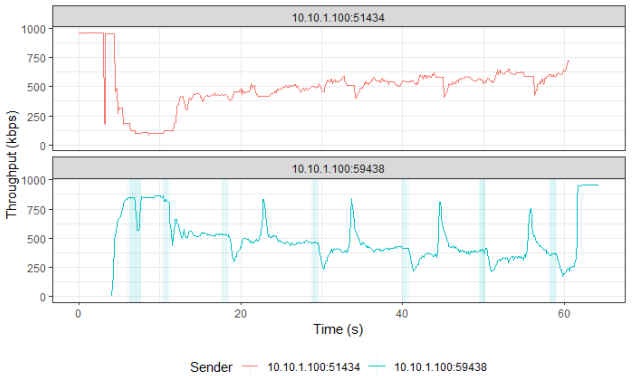


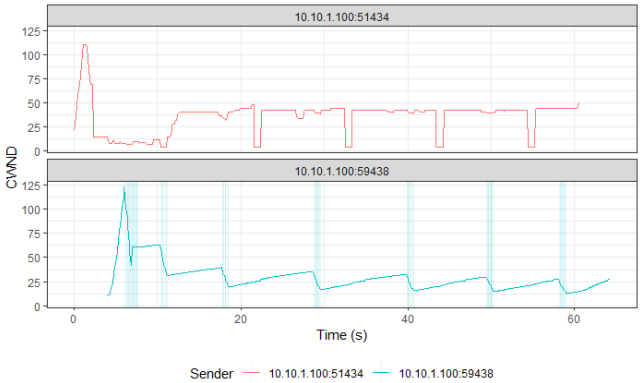
Figure credits: queue.acm.org/detail.cfm?id=3022184

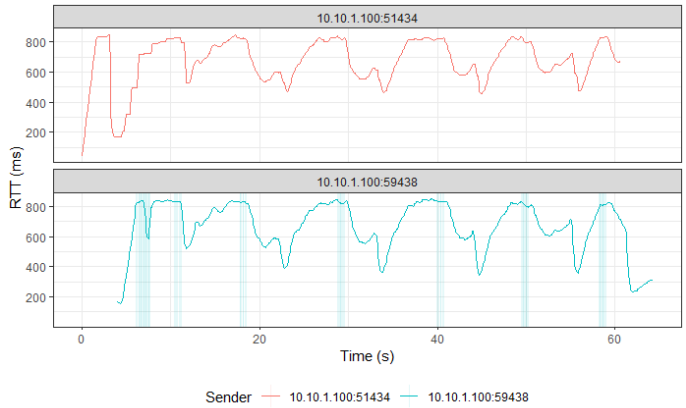
As TCP BBR is operating at a more optimal point than TCP Reno, so we expected BBR to outperform TCP Reno.

3.7.2. Result and Analysis

NOTE: TCP BBR followed by TCP Reno in all three graphs.







Although, TCP BBR's TPUT increases towards the end of the graph, over the entire connection, TCP Reno and TCP BBR were both able to use the connection fairly at ~500 Kbps each, as recorded by `iperf3`. This means that TCP BBR is able to hold its own in front of an aggressive algorithm like TCP Reno and it has its optimal point of operation and the consideration of both Bottleneck Bandwidth and RTT to decide the sending rate to thank.

One more interesting thing to note is that even with changes in RTT (as seen in the third graph in this section), TCP BBR is able to operate at a constant sending rate (as indicated by the steady CWND portion in the 2nd graph in this section), which is in contrast to TCP Reno's sending rate. This implies that the optimal point at which TCP BBR operates allows it to keep sending data at unchanged rates on a relatively congested link without getting bogged down by competing TCP flows such as TCP Reno.

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/reno-vs-bbr

Similar results were observed for TCP CUBIC vs TCP BBR, where TCP BBR was able to stand up to TCP CUBIC, much unlike TCP Vegas or TCP Reno.

More details:
github.com/HarshKapadia2/tcp-version-performance-comparison/tree/main/data/cubic-vs-bbr

4. Conclusion

4.1. Experiment No. 1

Learning about and comparing the Congestion Windows (CWND) of TCP Reno, TCP CUBIC, TCP Vegas and TCP BBR was revealing in terms of understanding how much difference different ways of thinking and algorithms can make to a protocol's functioning.

4.2. Experiment No. 2

Comparing the performance of TCP Reno, TCP CUBIC, TCP Vegas and TCP BBR was revealing in terms of how difficult it is to design TCP versions that are able to utilize the available bandwidth to its maximum and at the same time be fair to other TCP flows on the same connection.

TCP Reno and TCP CUBIC are at one end of the spectrum, where they utilize the available bandwidth to its maximum and reach the point of maximum utilization quickly, but are not fair to other TCP flows sharing the connection, whereas TCP Vegas is at the other end, being very civil to other TCP flows on the connection, but getting eaten up due to its uncompetitiveness.

TCP BBR was the only TCP version that was able to behave fairly with the aggressive TCP Reno and TCP CUBIC versions.

5. Extensions

A number of ideas can be further experimented with to draw more conclusions about the performance of the four TCP versions in this project and other TCP versions not in this project as well.

1. Change the delay on both the Sender (Romer) and Receiver (Juliet), and check the effect that has on the TPUT of each TCP version. This will show how different TCP versions respond to added delay, because some versions are loss-based, while some are delay-based.
2. Make the network lossy (different percentages) and check the TPUT and RTT for each TCP version.
3. Vary the buffer capacity (speed/rate) and check the TPUT and RTT for every TCP version.
4. Vary the buffer size (amount of bytes held in the buffer) and check the TPUT and RTT for every TCP version.

6. References

All references and study material listed here: networking.harshkapadia.me/tcp

7. Division of Labor

1. Experiment No. 1 execution: Yuxuan Huang and Haniel Edward Jacob Thomson
2. Experiment No. 2 execution: Harsh Gaurang Kapadia and Pranesh Jayasundar
3. Writeup: Harsh Gaurang Kapadia, Pranesh Jayasundar, Yuxuan Huang and Haniel Edward Jacob Thomson