UNIVERSITY OF
SOUTH CAROLINA

NETWORK TOOLS AND PROTOCOLS

Lab 7: Understanding Rate-based TCP Congestion Control (BBR)

Document Version: 06-14-2019

Award 1829698
"CyberTraining CIP: Cyberinfrastructure Expertise on High-throughput Networks for Big Science Data Transfers"

Contents

Overview

This lab describes a new type of TCP congestion control algorithm called Bottleneck Bandwidth and Round-Trip Time (BBR). The lab conducts experimental results using TCP BBR and contrasts these results with those obtained using traditional congestion control algorithms such as a Reno and HTCP.

Objectives

By the end of this lab, students should be able to:

1. Describe the basic operation of TCP BBR.
2. Describe differences between rate-based congestion control and window-based loss-based congestion control.
3. Modify the TCP congestion control algorithm in Linux using sysctl tool.
4. Compare the throughput performance of TCP Reno and BBR in high-throughput high-latency networks.

Lab settings

The information in Table 1 provides the credentials of the machine containing Mininet.

Table 1**.** Credentials to access Client1 machine.

Device	Account	Password
Client1	admin	password

Lab roadmap

This lab is organized as follows:

1. Section 1: Introduction to TCP.
2. Section 2: Lab Topology.
3. Section 3: iPerf3 Throughput Test.

1 Introduction to TCP

1.1 Traditional TCP congestion control review

TCP congestion control was introduced in the late 1980s. For many years, the main algorithm of congestion control was TCP Reno[1]. Subsequently, multiple algorithms were proposed based on Reno's enhancements. The goal of congestion control is to determine how much capacity is available in the network, so that a source knows how many packets it can safely have in transit (inflight). Once a source has these packets in transit, it uses the arrival of an acknowledgement (ACK) as a signal that one of its packets has left the network and that it is therefore safe to insert a new packet into the network without adding to the level of congestion. By using ACKs to pace the transmission of packets, TCP is said to be self-clocking[2].

A major task of the congestion control algorithm is to determine the available capacity. In steady state, TCP Reno maintains an estimate of the Round-Trip Time (RTT) -the time to send a packet and receive the corresponding ACK-. If the ACK stream shows that no packets are lost in transit, Reno increases the sending rate by one additional segment each RTT interval. This period is known as the additive increase. Note that "segment" here refers to the protocol data unit (PDU) at the transport layer, and that sometimes the terms packet and segment are interchangeably used. Eventually, the increasing flow rate saturates the bottleneck link at a router, which drops a packet. The TCP receiver signals the missing packet by sending an ACK in response to an out-of-order received segment, as illustrated in Figure 1(a). Once the TCP sender receives three duplicate ACKs for the same out-of-order segment, it interprets this event as packet loss due to congestion and reduces the sending rate by half. This reduction is known as multiplicative decrease. Once the loss is repaired, Reno resumes the additive increase phase. This iteration of additive increase multiplicative decrease (AIMD) periods is shown in Figure 1(b).

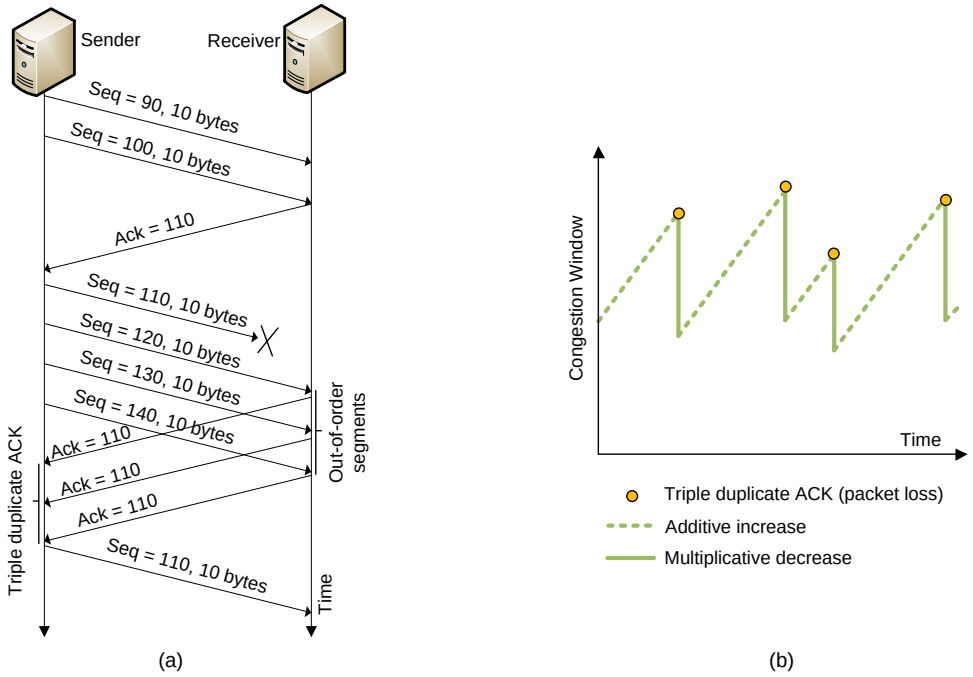

Figure 1. (a) TCP operation. (b) Evolution of TCP's congestion window.

1.2 Traditional congestion control limitations

While Reno has proven to perform adequately in the past, when the bulk of the TCP connections carried trivial applications such as web browsing and email, it faces severe limitations in high-throughput connections that are needed for grid computing and big science data transfers. Reno's average TCP throughput can be approximated by the following equation[2]:

$$\text{TCP Throughput} \approx \frac{\text{MSS}}{\text{RTT}\sqrt{L}} \quad \text{[bytes / second]}$$

The equation above indicates that the throughput of a TCP connection in steady state is directly proportional to the maximum segment size (MSS) and inversely proportional to the product of Round-Trip Time (RTT) and the square root of the packet loss rate (L). Essentially, the equation above indicates that the TCP throughput is very sensitive to packet loss. In such environments Reno cannot achieve high throughput, especially in high-latency scenarios. Figure 2 validates the above equation. It shows the throughput as a function of RTT, for two devices connected by a 10 Gbps path. The performance of two TCP AIMD-based implementations are provided: Reno[1] (blue) and Hamilton TCP[3], better known as HTCP (red). The theoretical performance (using the above equation) with packet losses (green) and the measured throughput without packet losses (purple) are also shown. Figure 2 is reproduced from[4].

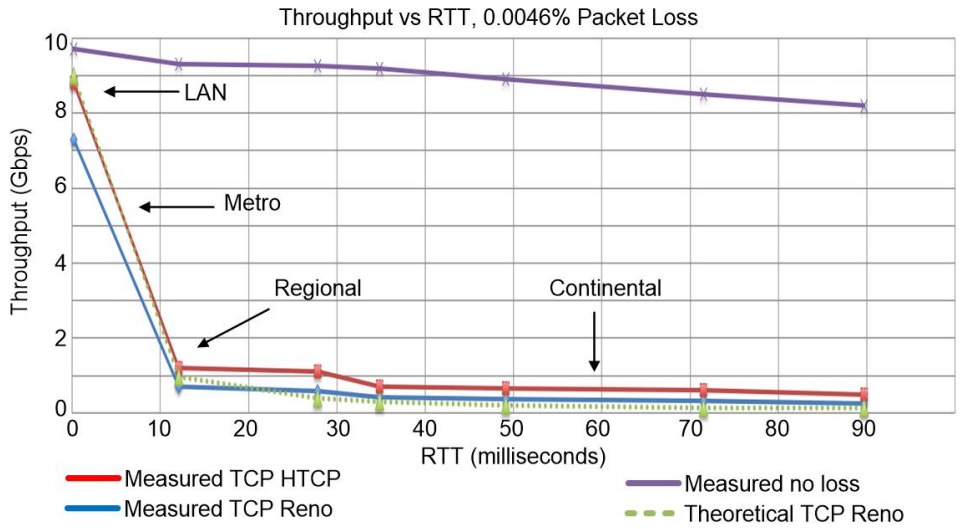

Figure 2. Throughput vs Round-Trip Time (RTT) for two devices connected via a 10 Gbps path. The performance of two TCP implementations are provided: Reno[1] (blue) and HTCP (red). The theoretical performance with packet losses (green) and the measured throughput without packet losses (purple) are also shown.

1.3 TCP BBR

The main issue surrounding traditional congestion control algorithms in high-speed high-latency networks is that the sender cannot recover from the packet loss and multiplicative decrease, even when the packet losses are sporadic. When the RTT is large, increasing the congestion window (and thus the sending rate) by only 1 MSS every RTT is too slow.

BBR[5] is a new congestion control algorithm that does not adhere to the AIMD rule and the above equation. BBR is a rate-based algorithm, meaning that at any given time it sends data at a rate that is independent of current packet losses. Note that this feature is a drastic departure from traditional congestion control algorithms, which operate by reducing the sending rate by half each time a packet loss is detected.

The behavior of BBR can be described using Figure 3, which shows a TCP's viewpoint of an end-to-end connection. At any time, the connection has exactly one slowest link, or bottleneck bandwidth (*btlbw*), that determines the location where queues are formed. When router buffers are large, traditional congestion control keeps them full (i.e., they keep increasing the rate during the additive increase phase). When buffers are small, traditional congestion control misinterprets a loss as a signal of congestion, leading to low throughput. The output port queue increases when the input link arrival rate exceeds *btlbw*. The throughput of loss-based congestion control is less than *btlbw* because of the frequent packet losses.

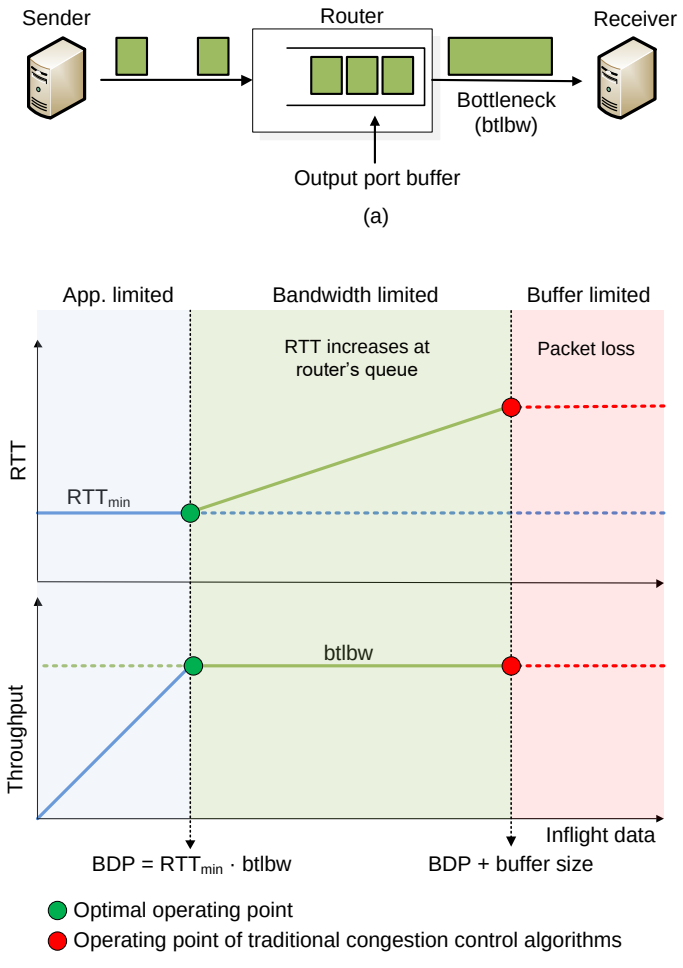

Figure 3. TCP viewpoint of a connection and relation between throughput and RTT. (a) Simplified TCP interpretation of the connection. (b) Throughput and RTT, as a function of in-flight data.

Figure 3(b) illustrates the RTT and throughput with the amount of data inflight[5]. RTT_{min} is the propagation time with no queueing component (the network is not congested). In the

application limited region, the delivery rate/throughput increases as the amount of data generated by the application layer increases, while the RTT remains constant. The pipeline between sender and receiver becomes full when the inflight number of bits is equal to the bandwidth multiplied by the RTT. This number is also called bandwidth-delay product (BDP) and quantifies the number of bits that can be inflight if the sender continuously sends segments. In the bandwidth limited region, the queue size at the router of Figure 3(a) starts increasing, resulting in an increase of the RTT. The throughput remains constant, as the bottleneck link is fully utilized. Finally, when no buffer is available at the router to store arriving packets (the number of inflight bits is equal to BDP plus the buffer size of the router), these are dropped.

It is important to understand that packets to be sent are paced at the estimated bottleneck rate, which is intended to avoid network queuing that would otherwise be encountered when the network performs rate adaptation at the bottleneck point. The intended operational model here is that the sender is passing packets into the network at a rate that is not anticipated to encounter queuing at any point within the entire path. This is a significant contrast to protocols such as Reno, which tends to send packet bursts at the epoch of the RTT and relies on the network's queues to perform rate adaptation in the interior of the network if the burst sending rate is higher than the bottleneck capacity.

BBR also periodically probes for additional bandwidth. It spends one RTT interval deliberately sending at a rate that is higher than the current estimate bottleneck bandwidth. Specifically, it sends data at 125% the bottleneck bandwidth. If the available bottleneck bandwidth has not changed, then the increased sending rate will cause a queue to form at the bottleneck. This will cause the ACK signaling to reveal an increased RTT, but the bottleneck bandwidth estimate will be unaltered. If this is the case, then the sender will subsequently send at a compensating reduced sending rate for an RTT interval. The reduced rate is set to 75% the bottleneck bandwidth, allowing the bottleneck queue to drain. On the other hand, if the available bottleneck bandwidth estimate has increased because of this probe, then the sender will operate according to this new bottleneck bandwidth estimate. The entire cycle duration lasts eight RTTs and is repeated indefinitely in steady state.

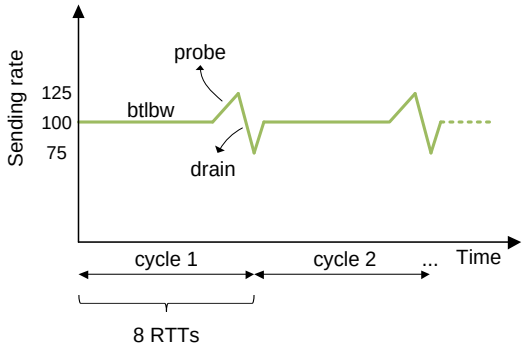

Figure 4. The rate used by the sender is the estimate bottleneck bandwidth (*btlbw*). During the probe period (1 RTT duration), the sender probes for additional bandwidth, sending at a rate of 125% of the bottleneck bandwidth. During the subsequent period, drain (1 RTT duration), the sender reduces the rate to 75% of the bottleneck bandwidth, thus allowing any bottleneck queue to drain.

2 Lab topology

Let's get started with creating a simple Mininet topology using MiniEdit. The topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

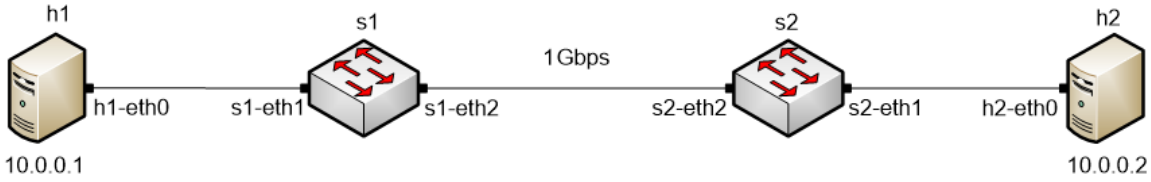

Figure 5. Lab topology.

Step 1. A shortcut to MiniEdit is located on the machine's Desktop. Start MiniEdit by clicking on MiniEdit's shortcut. When prompted for a password, type `password`.

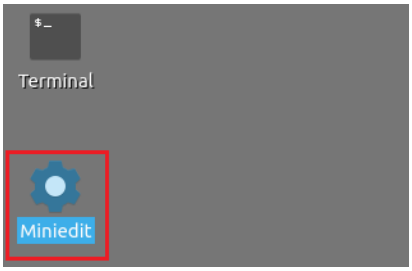

Figure 6. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. Locate the *Lab 7.mn* topology file and click on *Open*.

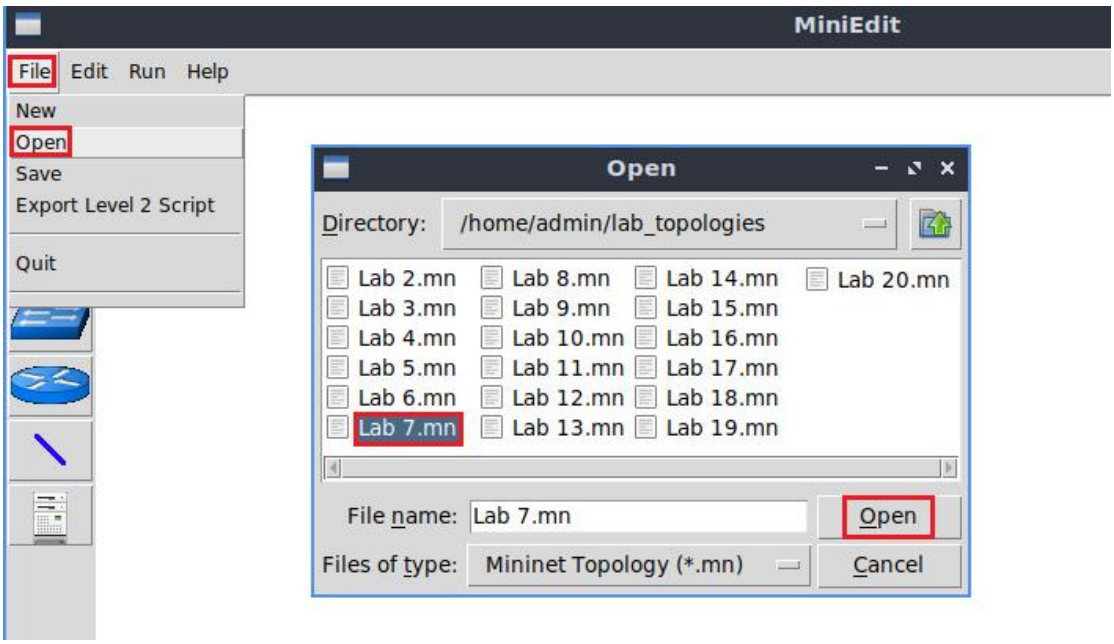

Figure 7. MiniEdit's *Open* dialog.

Step 3. Before starting the measurements between host h1 and host h2, the network must be started. Click on the *Run* button located at the bottom left of MiniEdit's window to start the emulation.

Figure 8. Running the emulation.

The above topology uses 10.0.0.0/8 which is the default network assigned by Mininet.

2.1 Starting host h1 and host h2

Step 1. Hold the right-click on host h1 and select *Terminal*. This opens the terminal of host h1 and allows the execution of commands on that host.

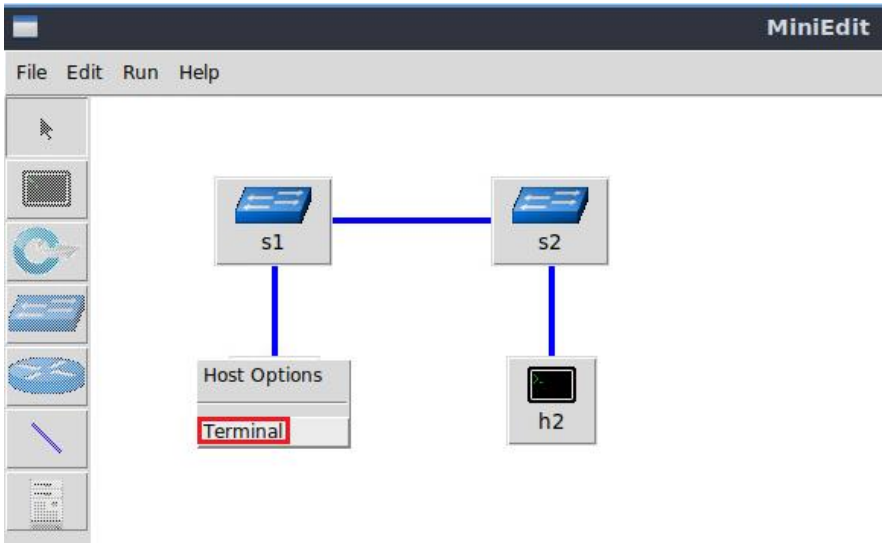

Figure 9. Opening a terminal on host h1.

Step 2. Apply the same steps on host h2 and open its *Terminal*.

Step 3. Test connectivity between the end-hosts using the `ping` command. On host h1, type the command `ping 10.0.0.2`. This command tests the connectivity between host h1 and host h2. To stop the test, press `Ctrl+c`. The figure below shows a successful connectivity test.

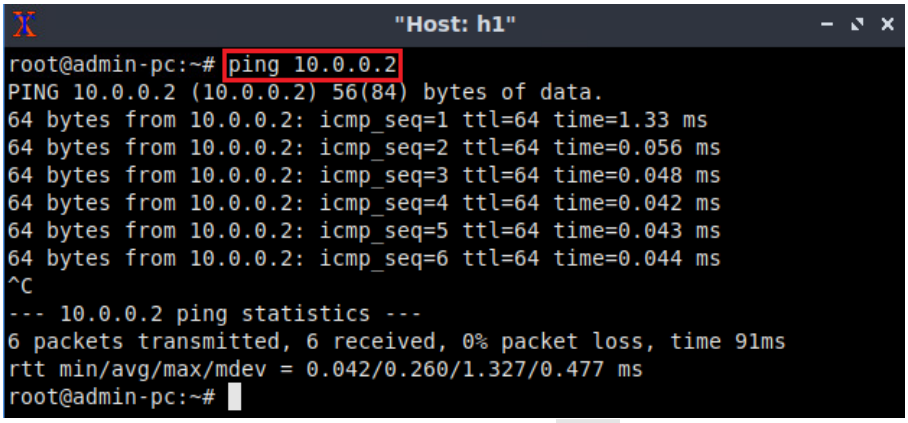

Figure 10. Connectivity test using `ping` command.

Figure 10 indicates that there is connectivity between host h1 and host h2. Thus, we are ready to start the throughput measurement process.

2.2 Emulating 1 Gbps high-latency WAN with packet loss

This section emulates a high-latency WAN, which is used to validate the results observed in Figure 3. We will first set the bandwidth between host h1 and host h2 to 1 Gbps. Then we will emulate packet losses between switch S1 and switch S2, and measure the throughput.

Step 1. Launch a Linux terminal by holding the `Ctrl+Alt+T` keys or by clicking on the Linux terminal icon.

Figure 11. Shortcut to open a Linux terminal.

The Linux terminal is a program that opens a window and permits you to interact with a command-line interface (CLI). A CLI is a program that takes commands from the keyboard and sends them to the operating system for execution.

Step 2. In the terminal, type the below command. When prompted for a password, type `password` and hit enter. This command basically introduces a 0.01% packet loss rate on switch S1's *s1-eth2* interface.

```
sudo tc qdisc add dev s1-eth2 root handle 1: netem loss 0.01%
```

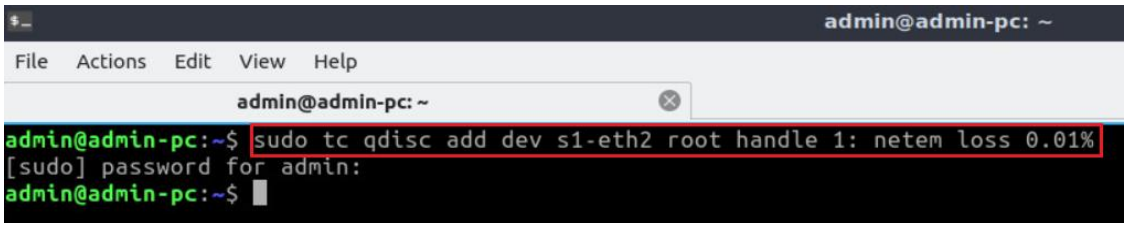

Figure 12. Adding 0.01% packet loss rate to switch S1's s1-eth2 interface.

Step 3. Modify the bandwidth of the link connecting the switch S1 and switch S2: on the same terminal, type the command below. This command sets the bandwidth to 1 Gbps on switch S1's *s1-eth2* interface. The `tbf` parameters are the following:

- `rate`: 1gbit
- `burst`: 500,000
- `limit`: 2,500,000

```
sudo tc qdisc add dev s1-eth2 parent 1: handle 2: tbf rate 1gbit burst 500000
limit 2500000
```

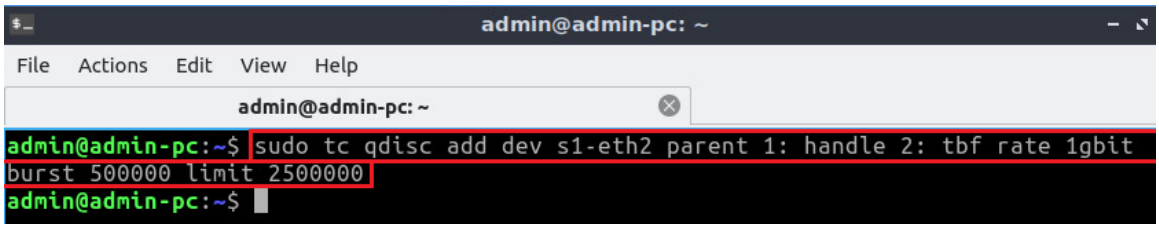

Figure 13. Limiting the bandwidth to 1 Gbps on switch S1's s1-eth2 interface.

2.3 Testing connection

To test connectivity, you can use the command `ping`.

Step 1. On the terminal of host h1, type `ping 10.0.0.2`. To stop the test, press `Ctrl+c`. The figure below shows a successful connectivity test. Host h1 (10.0.0.1) sent four packets to host h2 (10.0.0.2), successfully receiving responses back.

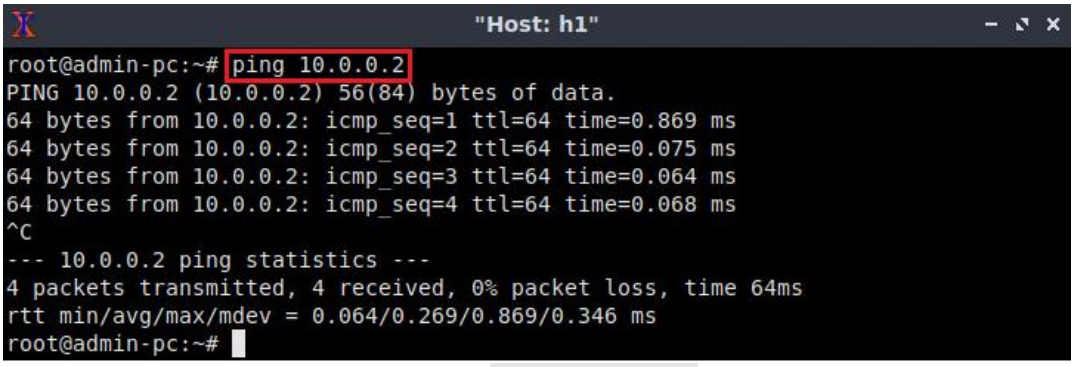

Figure 14. Output of `ping 10.0.0.2` command.

The result above indicates that all four packets were received successfully (0% packet loss) and that the minimum, average, maximum, and standard deviation of the Round-Trip Time (RTT) were 0.064, 0.269, 0.869, and 0.346 milliseconds, respectively. Essentially, the standard deviation is an average of how far each ping RTT is from the average RTT. The higher the standard deviation the more variable the RTT is.

Step 2. On the terminal of host h2, type `ping 10.0.0.1`. The ping output in this test should be relatively close to the results of the test initiated by host h1 in Step 1. To stop the test, press `Ctrl+c`.

3 iPerf3 throughput test

In this section, the throughput between host h1 and host h2 is measured using two congestion control algorithms: Reno and BBR. Moreover, the test is repeated using various injected delays to observe the throughput variations depending on each congestion control algorithm and the selected RTT.

3.1 Throughput test without delay

In this test, we measure the throughput between host h1 and host h2 without introducing delay on the switch S1's *s1-eth2* interface.

3.1.1 TCP Reno

Step 1. In host h1's terminal, change the TCP congestion control algorithm to Reno by typing the following command:

```
sysctl -w net.ipv4.tcp_congestion_control=reno
```

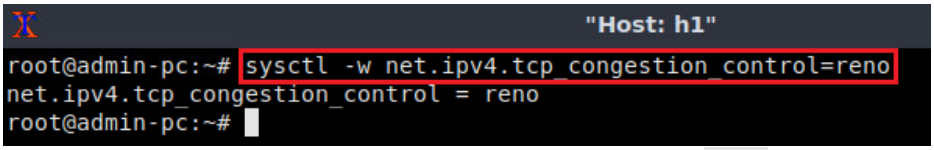

Figure 15. Changing TCP congestion control algorithm to `reno` on host h1.

Step 2. Launch iPerf3 in server mode on host h2's terminal:

```
iperf3 -s
```

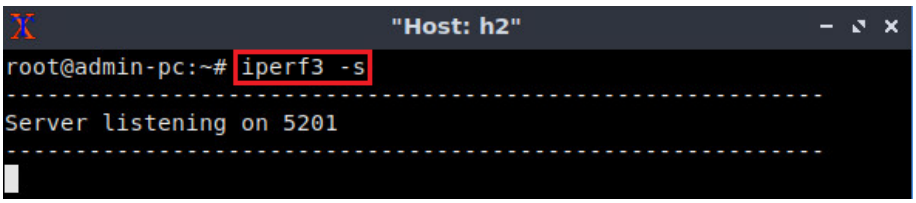

Figure 16. Starting iPerf3 server on host h2.

Step 3. Launch iPerf3 in client mode on host h1's terminal. The `-O` option is used to specify the number of seconds to omit in the resulting report.

```
iperf3 -c 10.0.0.2 -t 20 -O 10
```

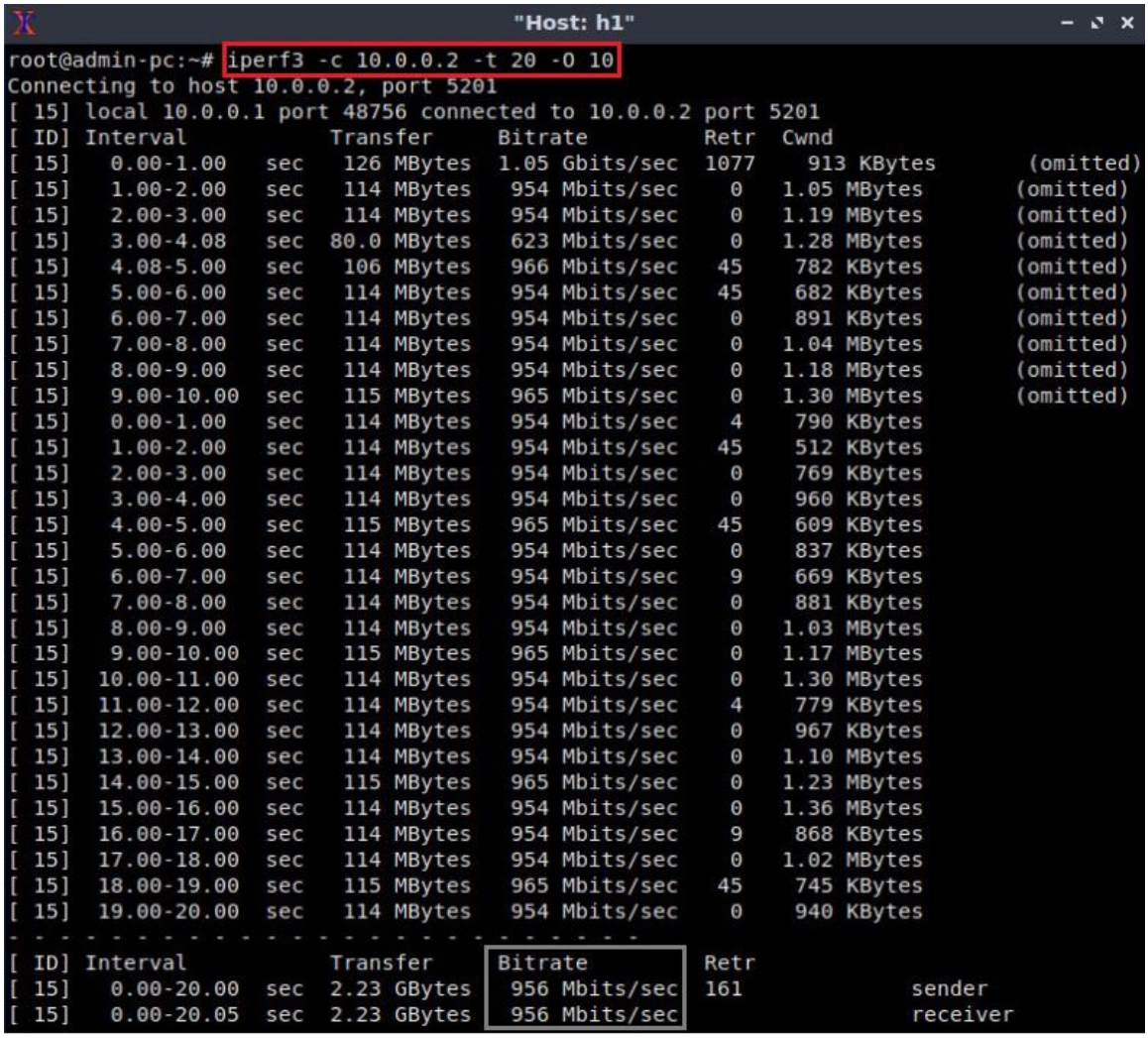

Figure 17. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughputs are 956 Mbps (sender) and 956 Mbps (receiver), and the number of retransmissions is 161 (due to the injected packet loss - 0.01%).

Step 4. In order to stop the server, press `Ctrl+c` in host h2's terminal. The user can see the throughput results in the server side too.

3.1.2 TCP BBR

Step 1. In host h1's terminal, change the TCP congestion control algorithm to BBR by typing the following command:

```
sysctl -w net.ipv4.tcp_congestion_control=bbr
```

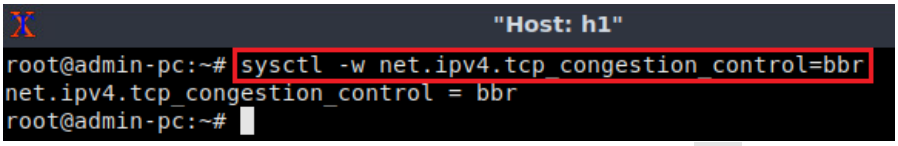

Figure 18. Changing TCP congestion control algorithm to `bbr` on host h1.

Step 2. Launch iPerf3 in server mode on host h2's terminal:

```
iperf3 -s
```

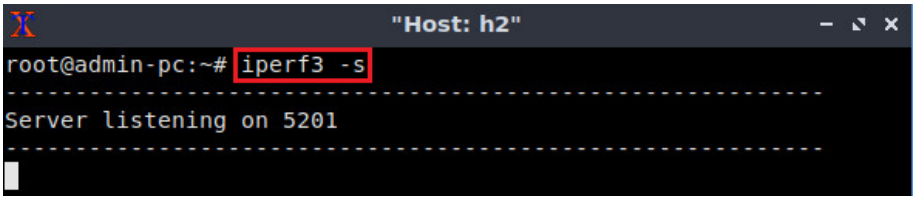

Figure 19. Starting iPerf3 server on host h2.

Step 3. Launch iPerf3 in client mode on host h1's terminal:

```
iperf3 -c 10.0.0.2 -t 20 -O 10
```

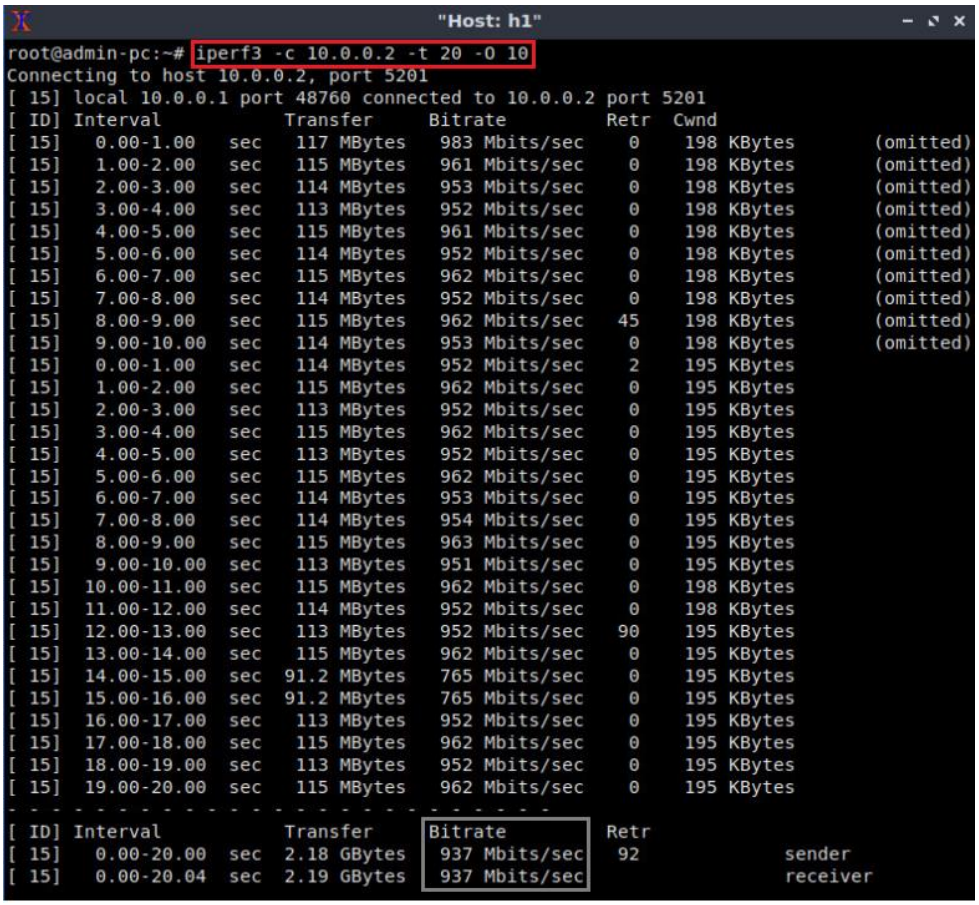

Figure 20. Running iPerf3 client on host h1.

Figure 20 shows the iPerf3 test output report. The average achieved throughputs are 937 Mbps (sender) and 937 Mbps (receiver), and the number of retransmissions is 92 (due to the injected packet loss - 0.01%).

Step 4. In order to stop the server, press `Ctrl+c` in host h2's terminal. The user can see the throughput results in the server side too.

3.2 Throughput test with 30ms delay

In this test, we measure the throughput between host h1 and host h2 while introducing 30ms delay on the switch S1's *s1-eth2* interface. Apply the following steps:

Step 1. In order to add delay to the switch 1 or interface s1-eth2, go back to the Client's terminal, run the following command to modify the previous rule to include 30ms delay:

```
sudo tc qdisc change dev s1-eth2 root handle 1: netem loss 0.01% delay 30ms
```

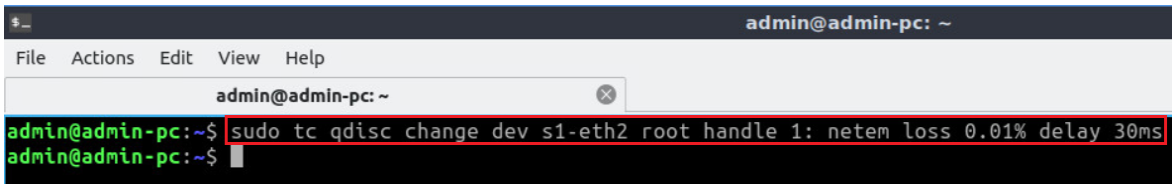

Figure 21. Injecting 30ms delay on switch S1's *s1-eth2* interface.

Step 2. In host h1's terminal, modify the TCP buffer size by typing the following commands: *sysctl -w net.ipv4.tcp_rmem='10,240 87,380 150,000,000'* and *sysctl -w net.ipv4.tcp_wmem='10,240 87,380 150,000,000'*. This TCP buffer is explained later in future labs.

```
sysctl -w net.ipv4.tcp_rmem='10240 87380 150000000'
```

```
sysctl -w net.ipv4.tcp_wmem='10240 87380 150000000'
```

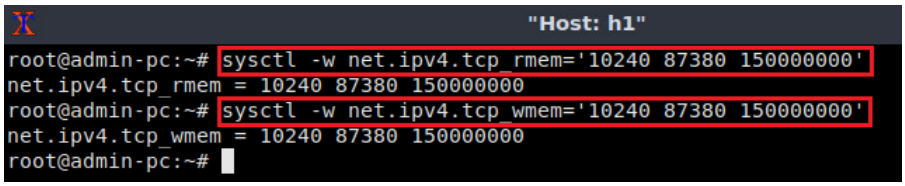

Figure 22. Modifying the TCP buffer size on host h1.

Step 3. In host h2's terminal, also modify the TCP buffer size by typing the following commands: *sysctl -w net.ipv4.tcp_rmem='10,240 87,380 150,000,000'* and *sysctl -w net.ipv4.tcp_wmem='10,240 87,380 150,000,000'*.

```
sysctl -w net.ipv4.tcp_rmem='10240 87380 150000000'
```

```
sysctl -w net.ipv4.tcp_wmem='10240 87380 150000000'
```

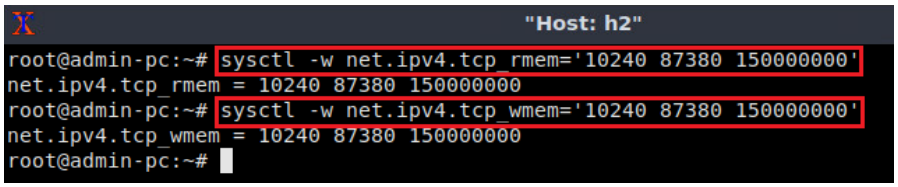

Figure 23. Modifying the TCP buffer size on host h2.

3.2.1 TCP Reno

Step 1. In host h1's terminal, change the TCP congestion control algorithm to Reno by typing the following command:

```
sysctl -w net.ipv4.tcp_congestion_control=reno
```

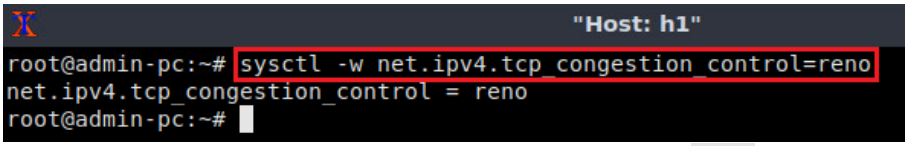

Figure 24. Changing TCP congestion control algorithm to `reno` on host h1.

Step 2. Launch iPerf3 in server mode on host h2's terminal:

```
iperf3 -s
```

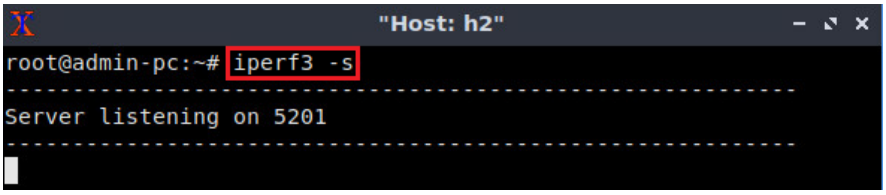

Figure 25. Starting iPerf3 server on host h2.

Step 3. Create and enter to a new directory *reno* on host h1's terminal:

```
mkdir reno && cd reno
```

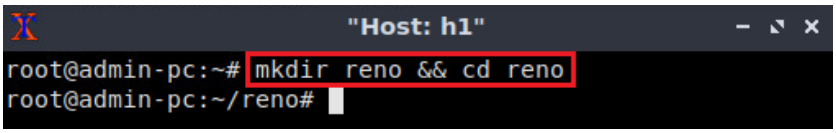

Figure 26. Creating and entering a new directory *reno*.

Step 4. Launch iPerf3 in client mode on host h1's terminal. The `-J` option is used to produce a JSON output and the redirection operator `>` to send the standard output to a file.

```
iperf3 -c 10.0.0.2 -t 30 -J > reno.json
```

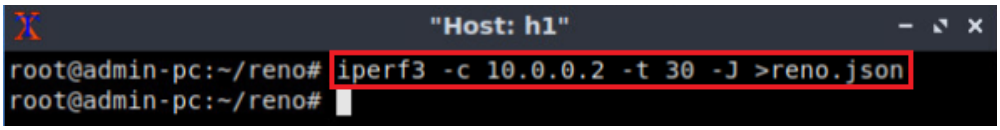

Figure 27. Running iPerf3 client on host h1 and redirecting the output to *reno.json*.

Step 5. Once the test is finished, type the following command to generate the output plots for iPerf3's JSON file:

```
plot_iperf.sh reno.json
```

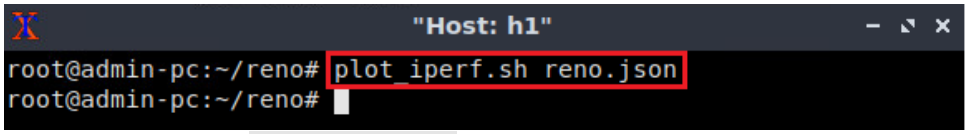

Figure 28. `plot_iperf.sh` script generating output results.

This plotting script generates PDF files for the following fields: congestion window (*cwnd.pdf*), retransmits (*retransmits.pdf*), Round-Trip Time (*RTT.pdf*), Round-Trip Time variance (*RTT_Var.pdf*), throughput (*throughput.pdf*), maximum transmission unit (*MTU.pdf*), bytes transferred (*bytes.pdf*). The plotting script also generates a CSV file (*1.dat*) to be used by applicable programs. These files are stored in a directory *results* created in the same directory where the script was executed as shown in the figure below.

Step 6. Navigate to the results folder using the `cd` command.

```
cd results/
```

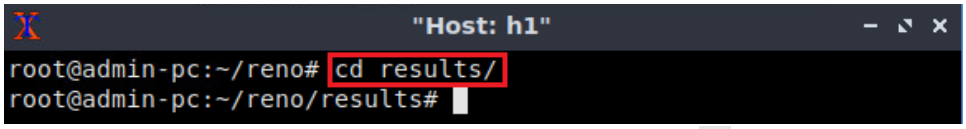

Figure 29. Entering the results directory using the `cd` command.

Step 7. To open any of the generated files, use the `xdg-open` command followed by the file name. For example, to open the *throughput.pdf* file, use the following command:

```
xdg-open throughput.pdf
```

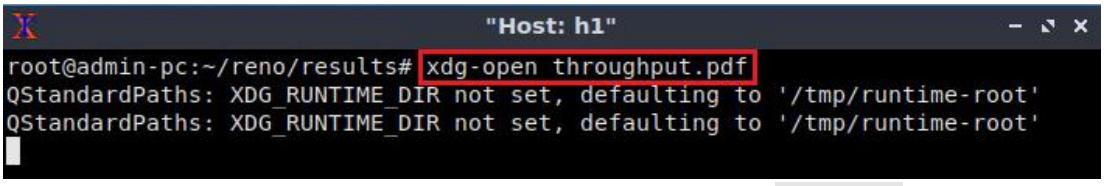

Figure 30. Opening the *throughput.pdf* file using `xdg-open`.

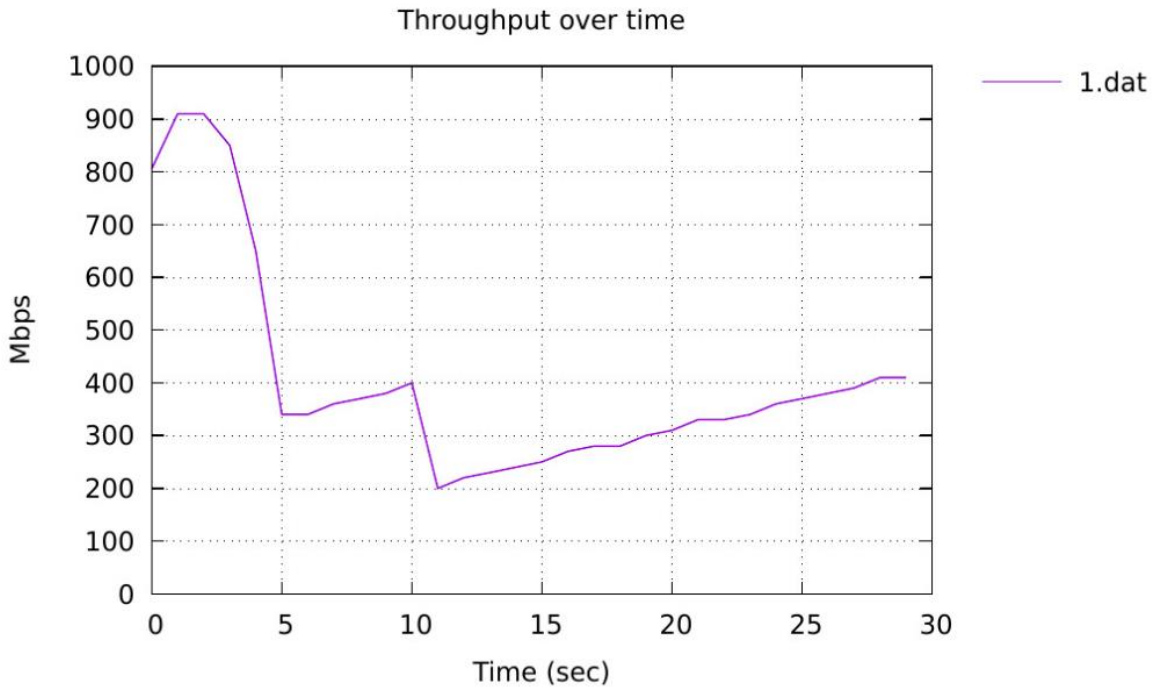

Figure 31. Reno's throughput.

Step 8. Close the *throughput.pdf* file and open the *cwnd.pdf* file using the following command:

```
xdg-open cwnd.pdf
```

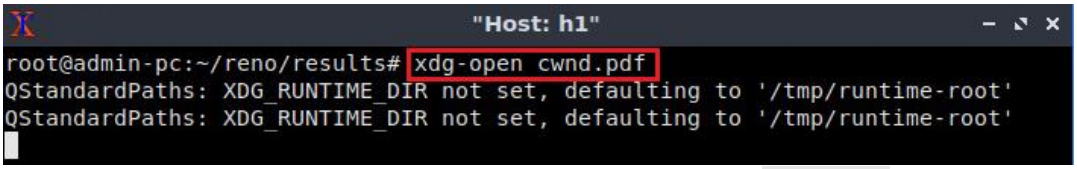

Figure 32. Opening the *throughput.pdf* file using `xdg-open`.

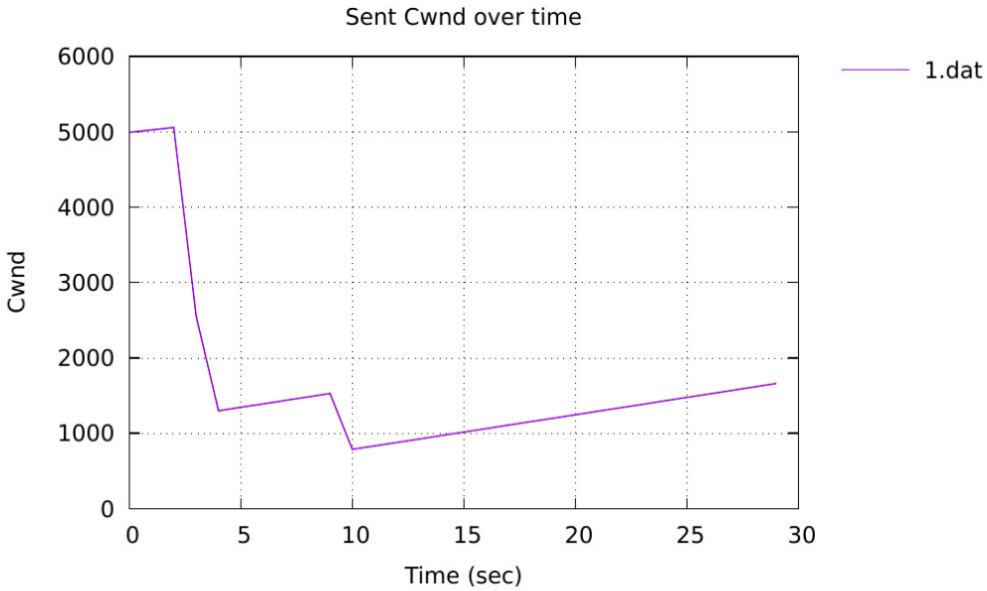

Figure 33. Reno's congestion window.

Step 9. In order to stop the server, press `Ctrl+c` in host h2's terminal. The user can see the throughput results in the server side too.

Step 10. Exit the */reno/results* directory by using the following command on host h1's terminal:

```
cd ../..
```

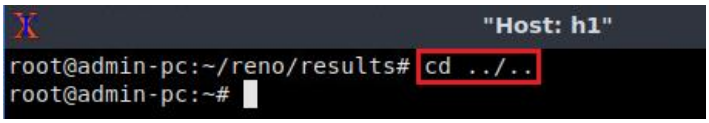

Figure 34. Exiting the */reno/results* directory.

3.2.2 TCP BBR

Step 1. In host h1's terminal, change the TCP congestion control algorithm to BBR by typing the following command:

```
sysctl -w net.ipv4.tcp_congestion_control=bbr
```

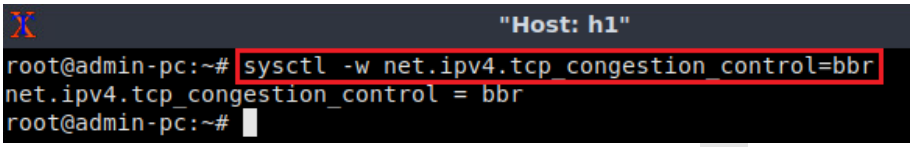

Figure 35. Changing TCP congestion control algorithm to `bbr` on host h1.

Step 2. Launch iPerf3 in server mode on host h2's terminal:

```
iperf3 -s
```

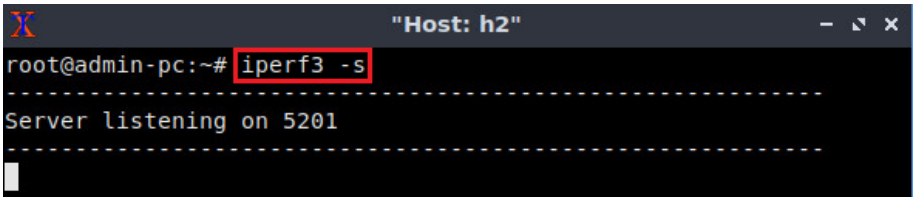

Figure 36. Starting iPerf3 server on host h2.

Step 3. Create and enter to a new directory *bbr* host h1's terminal*:*

```
mkdir bbr && cd bbr
```

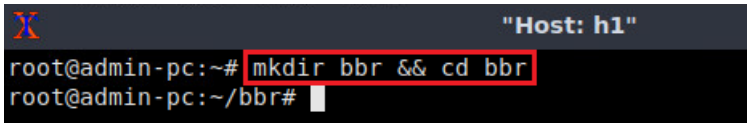

Figure 37. Creating and entering a new directory *bbr* .

Step 4. Launch iPerf3 in client mode on host h1's terminal. The `-J` option is used to produce a JSON output and the redirection operator `>` to send the standard output to a file.

```
iperf3 -c 10.0.0.2 -t 30 -J > bbr.json
```

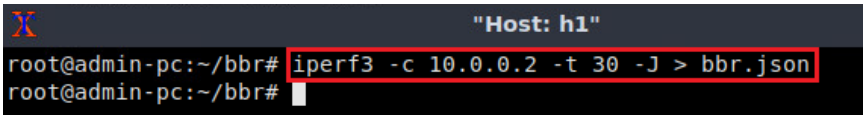

Figure 38. Running iPerf3 client on host h1 and redirecting the output to *bbr.json*.

Step 5. To generate the output plots for iPerf3's JSON file run the following command:

```
plot_iperf.sh bbr.json
```

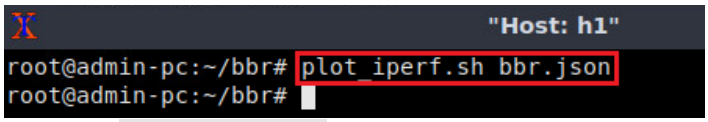

Figure 39. `plot_iperf.sh` script generating output results.

This plotting script generates PDF files for the following fields: congestion window (*cwnd.pdf*), retransmits (*retransmits.pdf*), Round-Trip Time (*RTT.pdf*), Round-Trip Time variance (*RTT_Var.pdf*), throughput (*throughput.pdf*), maximum transmission unit (*MTU.pdf*), bytes transferred (*bytes.pdf*). The plotting script also generates a CSV file (*1.dat*) to be used by applicable programs. These files are stored in a directory *results* created in the same directory where the script was executed as shown in the figure below.

Step 6. Navigate to the results folder using the `cd` command.

```
cd results/
```

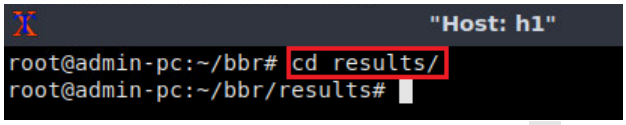

Figure 40. Entering the results directory using the `cd` command.

Step 7. To open any of the generated files, use the `xdg-open` command followed by the file name. For example, to open the *throughput.pdf* file, use the following command:

```
xdg-open throughput.pdf
```

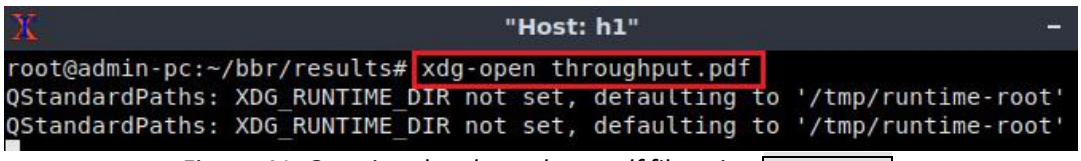

Figure 41. Opening the *throughput.pdf* file using `xdg-open`.

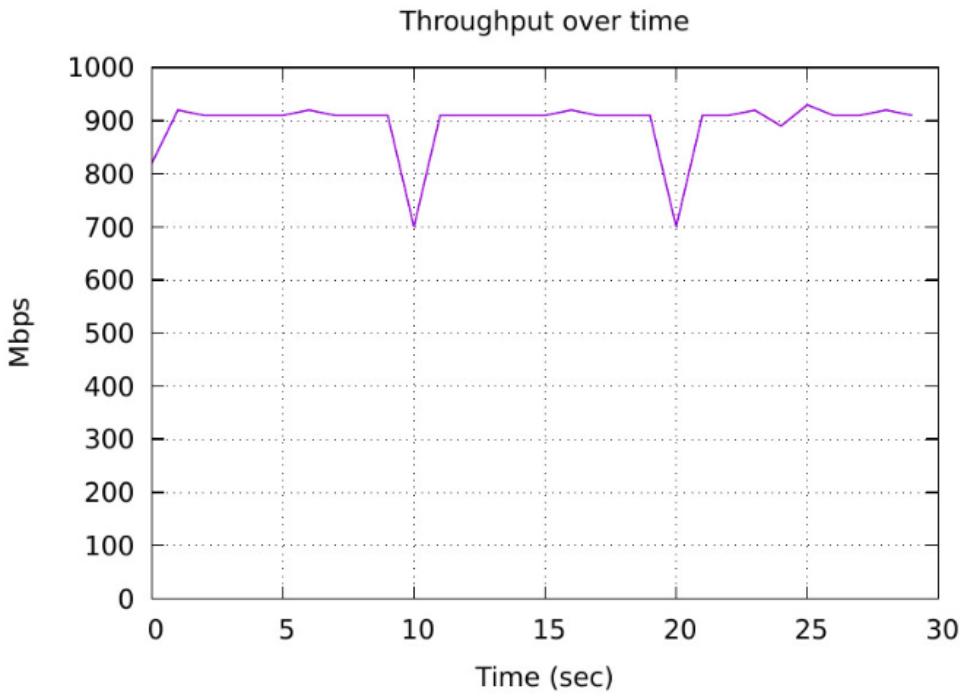

Figure 42. BBR's throughput.

Step 8. Figure 42 shows that in steady state, BBR has already attained the maximum throughput, which is over 900 Mbps (the bottleneck bandwidth is 1 Gbps, with an observed effective bandwidth of ~937 Gbps). Note also the periodic (short) drain intervals, where the throughput decreases to ~75% of maximum throughput, as discussed in Section 1.3. To proceed, close the *throughput.pdf* file and open the *cwnd.pdf* file using the following command:

```
xdg-open cwnd.pdf
```

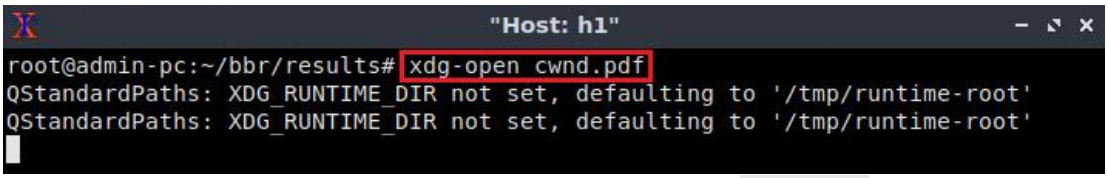

Figure 43. Opening the *cwnd.pdf* file using `xdg-open`.

Figure 44. BBR's congestion window.

Step 9. In order to stop the server, press `Ctrl+c` in host h2's terminal. The user can see the throughput results in the server side too.

Step 10. Exit the */bbr/results* directory by using the following command on host h1's terminal:

```
cd ../..
```

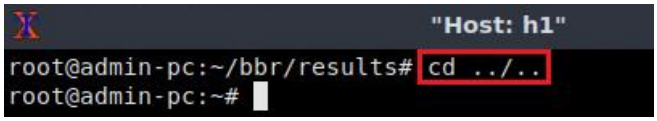

Figure 45. Exiting the */bbr/results* directory.

It is clear from the above test that when introducing delay, BBR preforms significantly better than Reno.

This concludes Lab 7. Stop the emulation and then exit out of MiniEdit.

References

1. K. Fall, S. Floyd, "Simulation-based comparisons of tahoe, reno, and sack TCP," Computer Communication Review, vol. 26, issue 3, Jul. 1996.
2. J. Kurose, K. Ross, "Computer networking, a top down approach," Pearson, 6th Edition, 2017.
3. D. Leith, R. Shorten, Y. Lee, "H-TCP: a framework for congestion control in high-speed and long-distance networks," Hamilton Institute Technical Report, Aug. 2005. [Online]. Available: http://www.hamilton.ie/net/htcp2005.pdf.
4. E. Dart, L. Rotman, B. Tierney, M. Hester, J. Zurawski, "The science DMZ: a network design pattern for data-intensive science," in Proceedings of the International

Conference on High Performance Computing, Networking, Storage and Analysis, Nov. 2013.
5. N. Cardwell, Y. Cheng, C. Gunn, S. Yeganeh, V. Jacobson, “BBR: Congestion-based congestion control,” Communications of the ACM, vol 60, no. 2, pp. 58-66, Feb. 2017.
6. S. Ha, I., Rhee, L. Xu, “CUBIC: a new TCP-friendly high-speed TCP variant,” ACM SIGOPS operating systems review, vol. 42, issue 5, pp. 64-74, Jul. 2008.
7. Leith D, Shorten R. H-TCP: TCP congestion control for high bandwidth-delay product paths. draft-leith-tcp-htcp-06 (work in progress). 2008 Apr.
8. System information variables – sysctl(7). [Online]. Available: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt.