UNIVERSITY OF
SOUTH CAROLINA

NETWORK TOOLS AND PROTOCOLS

Lab 6: Understanding Traditional TCP Congestion Control (HTCP, Cubic, Reno)

Document Version: 06-14-2019

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

Contents

Overview 3
Objectives........ 3
Lab settings 3
Lab roadmap 3
1 Introduction to TCP 3
1.1 TCP review 4
1.2 TCP throughput 4
1.3 TCP packet loss event 5
1.4 Impact of packet loss in high-latency networks........ 6
2 Lab topology........ 7
2.1 Starting host h1 and host h2 8
2.2 Emulating 10 Gbps high-latency WAN with packet loss 9
2.3 Testing connection 10
3 Introduction to sysctl 11
3.1 Read sysctl parameters 11
3.2 Write sysctl parameters 12
3.3 Configuring sysctl.conf file 12
4 Congestion control algorithms and sysctl........ 14
4.1 Inspect and install/load congestion control algorithms 15
4.2 Inspect the default (current) congestion control algorithm 16
4.3 Modify the default (current) congestion control algorithm 17
5 iPerf3 throughput test 17
5.1 Throughput test without delay 18
5.1.1 TCP Reno........ 18
5.1.2 Hamilton TCP (HTCP) 19
5.1.3 TCP Cubic 21
5.2 Throughput test with 30ms delay 22
5.2.1 TCP Reno........ 23
5.2.2 Hamilton TCP (HTCP) 24
5.2.3 TCP Cubic 26
References 27

Overview

This lab reviews key features and behavior of Transmission Control Protocol (TCP) that have a large impact on data transfers over high-throughput, high-latency networks. The lab describes the behavior of TCP's congestion control algorithm, its impact on throughput, and how to modify the congestion control algorithm in a Linux machine.

Objectives

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

1. Describe the basic operation of TCP congestion control algorithm and its impact on high-throughput networks.
2. Explain the concepts of congestion window, bandwidth probing, and Additive-Increase Multiplicative-Decrease (AIMD).
3. Understand TCP throughput calculation.
4. Understand the impact of packet loss on high-latency networks.
5. Deploy emulated WANs in Mininet.
6. Modify the TCP congestion control algorithm in Linux using *sysctl* tool.
7. Compare TCP Reno, HTCP, and Cubic with injected packet loss.
8. Compare TCP Reno, HTCP, and Cubic with both injected delay and packet loss.

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: Introduction to *sysctl*.
4. Section 4: Congestion control algorithms and *sysctl*.
5. Section 5: iPerf3 throughput test.

1 Introduction to TCP

1.1 TCP review

Big data applications require the transmission of large amounts of data between end devices. Data must be correctly delivered from one device to another; e.g., from an instrument to a Data Transfer Node (DTN). Reliability is one of the services provided by TCP and a reason why TCP is the protocol used by most data transfer tools. Thus, understanding the behavior of TCP is essential for the design and operation of networks used to transmit big data.

TCP receives data from the application layer and places it in the TCP send buffer, as shown in Figure 1(a). Data is typically broken into Maximum Segment Size (MSS) units. Note that "segment" here refers to the Protocol Data Unit (PDU) at the transport layer, and sometimes the terms packet and segment are interchangeably used. The MSS is simply the Maximum Transmission Unit (MTU) minus the combined lengths of the TCP and IP headers (typically 40 bytes). Ethernet's normal MTU is 1,500 bytes. Thus, MSS's typical value is 1,460. The TCP header is shown in Figure 1(b).

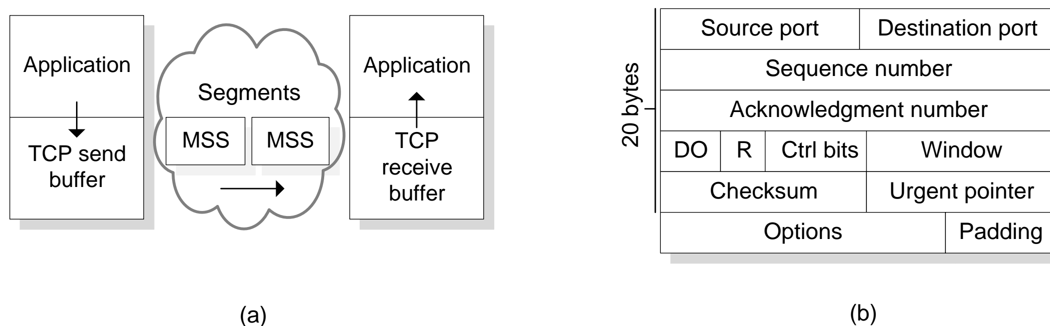

Figure 1. (a) TCP Connection. (b) TCP header.

For reliability, TCP uses two fields of the TCP header to convey information to the sender: sequence number and acknowledgement (ACK) number. The sequence number is the byte-stream number of the first byte in the segment. The acknowledgement number that the receiver puts in its segment is the sequence number of the next byte the receiver is expecting from the sender. In the example of Figure 2(a), after receiving the first two segments containing sequence number 90 (which contains bytes 90-99) and 100 (bytes 100-109), the receiver sends a segment with acknowledge number 110. This segment is called cumulative acknowledgement.

1.2 TCP throughput

The TCP rate limitation is defined by the receive buffer shown in Figure 1(a). If this buffer size is too small, TCP must constantly wait until an acknowledgement arrives before sending more segments. This limitation is removed by setting a large receive buffer size.

A second limitation is imposed by the congestion control mechanism operating at the sender side, which keeps track of a variable called congestion window. The congestion

window, referred to as *cwnd* (in bytes), imposes a constraint on the rate at which a TCP sender can send traffic. The *cwnd* value is the amount of unacknowledged data at the sender. To see this, note that at the beginning of every Round-Trip Time (RTT), the sender can send *cwnd* bytes of data into the connection; at the end of the RTT the sender receives acknowledgments for the data. Thus, the sender's send rate is roughly *cwnd*/RTT bytes/sec. By adjusting the value of *cwnd*, the sender can therefore adjust the rate at which it sends data into the connection.

$$\text{TCP Throughput} \approx \frac{\text{cwnd}}{\text{RTT}} \text{ [bytes/second]}$$

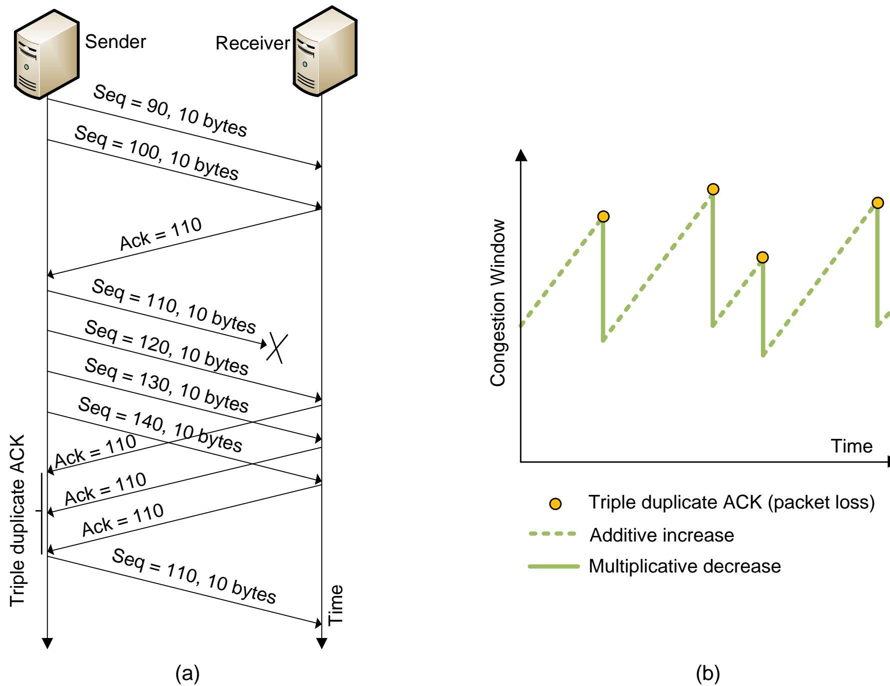

Figure 2. (a) TCP operation. (b) Adaptation of TCP's congestion window.

1.3 TCP packet loss event

TCP is a reliable transport protocol that requires each segment be acknowledged. If an acknowledgement for an outstanding segment is not received, TCP retransmits that segment. Alternatively, instead of waiting for a timeout-triggered retransmission, the sender can also detect a packet loss before the timeout by detecting duplicate ACKs. A duplicate ACK is an ACK that re-acknowledges a segment for which the sender has already received. If the TCP sender receives three duplicate ACKs for the same segment, TCP interprets this event as packet loss due to congestion and reduces the congestion window *cwnd* by half. This congestion window reduction is known as multiplicative decrease.

In steady state (ignoring the initial TCP period when a connection begins), a packet loss will be detected by a triple duplicate ACK. After decreasing *cwnd* by half, and as long as no other packet loss is detected, TCP will slowly increase *cwnd* again by 1 MSS per RTT. This congestion control phase essentially produces an additive increase in the congestion window. For this reason, TCP congestion control is referred to as an Additive-Increase Multiplicative-Decrease (AIMD) form of congestion control. AIMD gives rise to the "saw

tooth" behavior shown in Figure 2(b), which also illustrates the idea of TCP "probing" for bandwidth—TCP linearly increases its congestion window size (and hence its transmission rate) until a triple duplicate-ACK event occurs. It then decreases its congestion window size by a factor of two but then again begins increasing it linearly, probing to see if there is additional available bandwidth.

1.4 Impact of packet loss in high-latency networks

During the additive increase phase, TCP only increases *cwnd* by 1 MSS every RTT period. This feature makes TCP very sensitive to packet loss on high-latency networks, where the RTT is large.

Consider Figure 3, which shows the TCP throughput of a data transfer across a 10 Gbps path. The packet loss rate is 1/22,000, or 0.0046%. The purple curve is the throughput in a loss-free environment; the green curve is the theoretical throughput computed according to the equation below, where *L* is the packet loss rate.

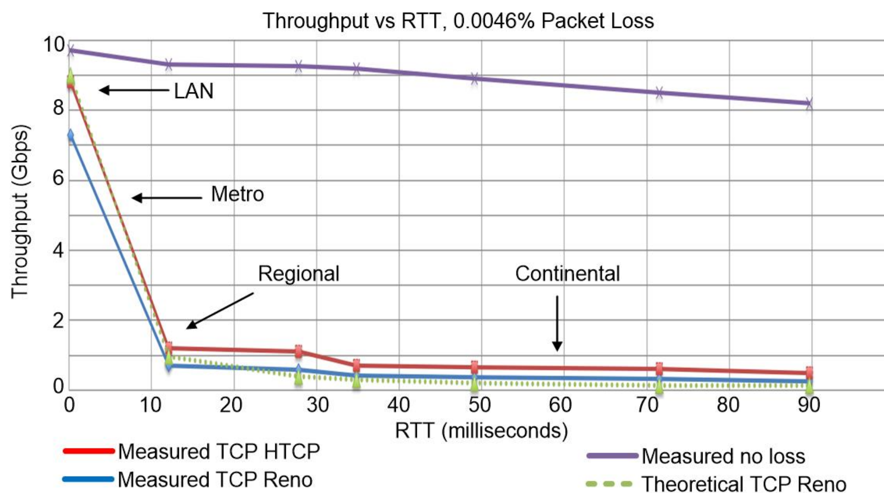

Figure 3. Throughput vs Round-Trip Time (RTT), for two devices connected via a 10 Gbps path. The performance of two TCP implementations are provided: Reno[1] (blue) and Hamilton TCP[2] (HTCP) (red). The theoretical performance with packet losses (green) and the measured throughput without packet losses (purple) are also shown[3].

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

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 Round-Trip Time (RTT) and the square root of the packet loss rate (*L*). The red and blue curves are real throughput measurements of two popular implementations of TCP: Reno[1] and Hamilton TCP (HTCP)[2]. Because TCP interprets losses as network congestion, it reacts by decreasing the rate at which packets are sent. This problem is exacerbated as the latency increases between the communicating hosts. Beyond LAN transfers, the throughput decreases rapidly to less than 1 Gbps. This is often the case when research collaborators sharing data are geographically distributed.

TCP Reno is an early congestion control algorithm. TCP Cubic[4], HTCP[5], and BBR[6] are more recent congestion control algorithms, which have demonstrated improvements with respect to TCP Reno.

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.

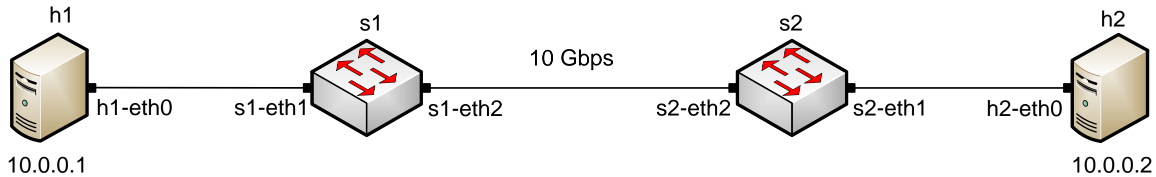

Figure 4. Lab topology.

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`.

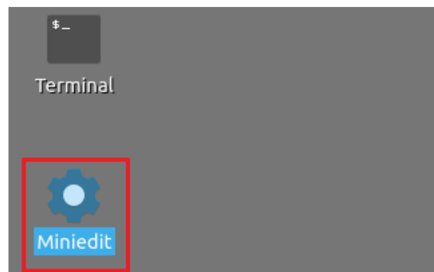

Figure 5. MiniEdit shortcut.

Step 2. On MiniEdit's menu bar, click on *File* then *Open* to load the lab's topology. Locate the *Lab 6.mn* topology file and click on *Open*.

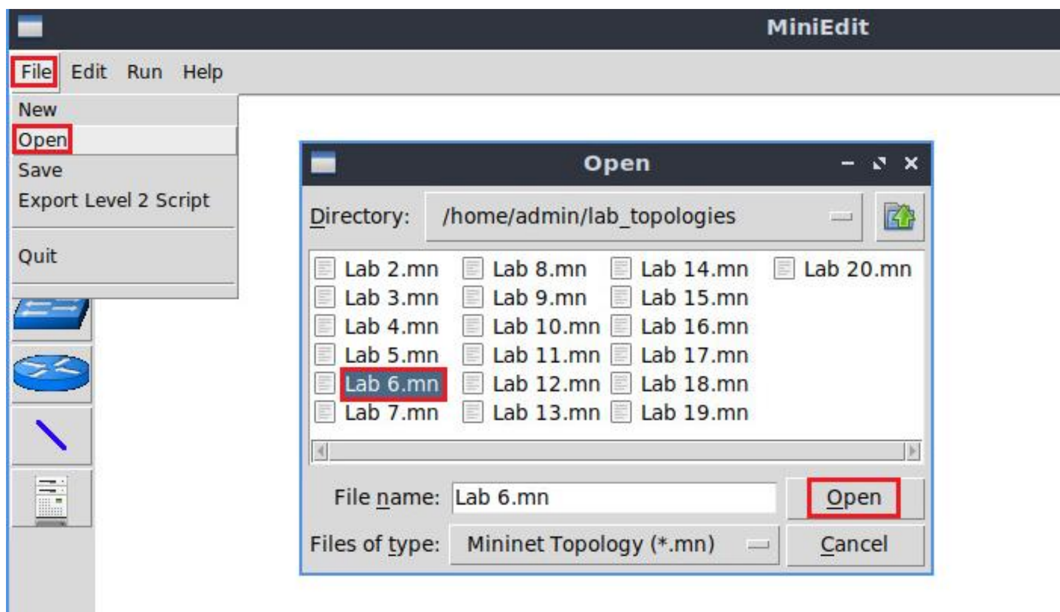

Figure 6. MiniEdit shortcut.

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 7. 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 host h1.

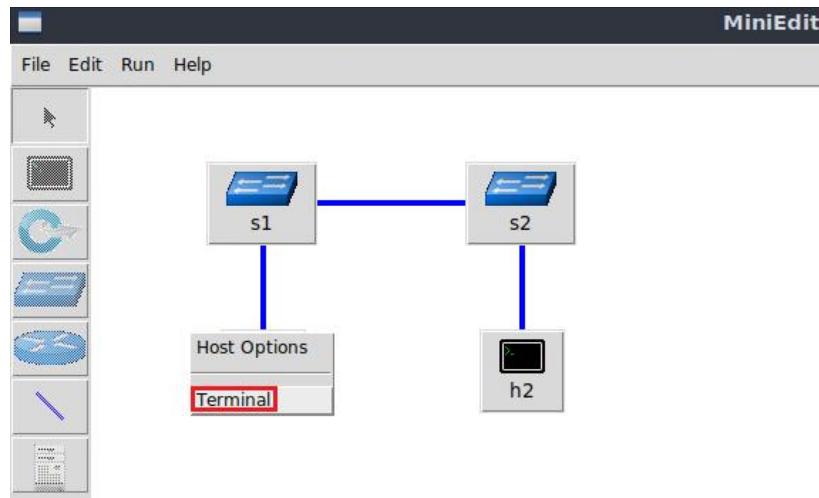

Figure 8. Opening a terminal on host h1.

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.

```
"Host: h1"
root@admin-pc:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=1.33 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.056 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.048 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.042 ms
64 bytes from 10.0.0.2: icmp_seq=5 ttl=64 time=0.043 ms
64 bytes from 10.0.0.2: icmp_seq=6 ttl=64 time=0.044 ms
^C
--- 10.0.0.2 ping statistics ---
6 packets transmitted, 6 received, 0% packet loss, time 91ms
rtt min/avg/max/mdev = 0.042/0.260/1.327/0.477 ms
root@admin-pc:~# 
```

Figure 9. Connectivity test using `ping` command.

Figure 9 indicates that there is connectivity between host h1 and host h2. Thus, we are ready to start the throughput measurement process.

2.2 Emulating 10 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 10 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.

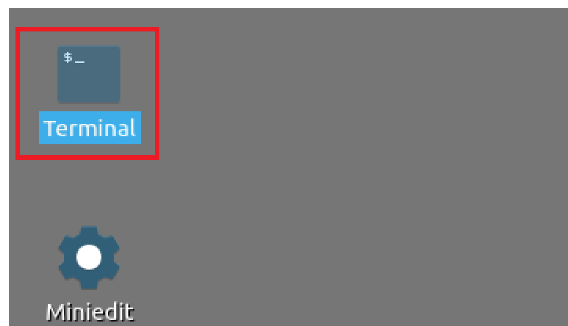

Figure 10. Shortcut to open a Linux terminal.

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 to perform.

Step 2. In the terminal, type the command below. When prompted for a password, type `password` and hit enter.

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

```
admin@admin-pc:~$ sudo tc qdisc add dev s1-eth2 root handle 1: netem loss 0.01%
[sudo] password for admin:
admin@admin-pc:~$
```

Figure 11. 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 10 Gbps on switch S1's *s1-eth2* interface. The `tbf` parameters are the following:

- `rate`: 10gbit
- `burst`: 5,000,000
- `limit`: 15,000,000

```
sudo tc qdisc add dev s1-eth2 parent 1: handle 2: tbf rate 10gbit burst 5000000
limit 15000000
```

```
admin@admin-pc:~$ sudo tc qdisc add dev s1-eth2 parent 1: handle 2: tbf rate 10gbit
burst 5000000 limit 15000000
admin@admin-pc:~$
```

Figure 12. Limiting the bandwidth to 10 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.

```
root@admin-pc:~# ping 10.0.0.2
PING 10.0.0.2 (10.0.0.2) 56(84) bytes of data.
64 bytes from 10.0.0.2: icmp_seq=1 ttl=64 time=0.869 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=0.075 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=0.064 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=0.068 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 64ms
rtt min/avg/max/mdev = 0.064/0.269/0.869/0.346 ms
root@admin-pc:~#
```

Figure 13. 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 similar to the results of the test initiated by host h1 in Step 1. To stop the test, press `Ctrl+c`.

3 Introduction to sysctl

sysctl is a tool for dynamically changing parameters in the Linux operating system[7]. It allows users to modify kernel parameters dynamically without rebuilding the Linux kernel.

Step 1. Run the command below on the Client1's terminal. When prompted for a password, type `password` and hit enter.

```
sudo sysctl -a
```

```
admin@admin-pc: ~
File  Actions  Edit  View  Help
admin@admin-pc: ~
admin@admin-pc:~$ sudo sysctl -a
[sudo] password for admin:
abi.vsyscall32 = 1
debug.exception-trace = 1
debug.kprobes-optimization = 1
dev.cdrom.autoclose = 1
dev.cdrom.autoeject = 0
dev.cdrom.check_media = 0
dev.cdrom.debug = 0
dev.cdrom.info = CD-ROM information, Id: cdrom.c 3.20 2003/12/17
dev.cdrom.info =
dev.cdrom.info = drive name:            sr0
```

Figure 14. Listing all system parameters in Linux.

This command produces a large output containing the kernel parameters and their values. This is represented in a key-value pair. For instance, `net.ipv4.ip_forward = 0` implies that the key `net.ipv4.ip_forward` has the value `0`.

3.1 Read sysctl parameters

It is often useful to search for specific keys without having to manually locate the needed key. This can be achieved using the following command:

```
sysctl <key>
```

Where *<key>* is replaced by the needed key. For example, the command `sysctl net.ipv4.ip_forward` returns `net.ipv4.ip_forward = 0`.

Step 1. Run the following command on the host h1's terminal:

```
sysctl net.ipv4.ip_forward
```

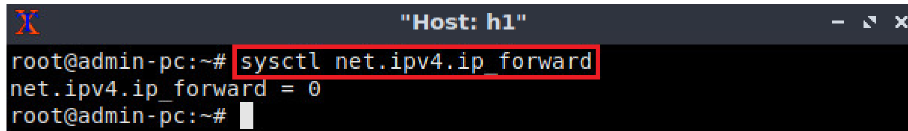

Figure 15. Reading the value of a given key.

3.2 Write sysctl parameters

It is also very useful to modify kernel parameters on the fly. The `-w` switch is added to the sysctl to "write" a value for a specific key.

```
sysctl -w <key>=<value>
```

Step 1. For example, if the user decides to enable IP forwarding (i.e., to configure a device as a router), then the following command is used:

```
sudo sysctl -w net.ipv4.ip_forward=1
```

Run the above command on the host h1's terminal:

Figure 16. Modifying a system parameter.

The changes made to a parameter using this command are temporary. Therefore, a new boot resets the value of a key to its default value. Also, when stopping MiniEdit's emulation, the configured parameters are reset.

3.3 Configuring sysctl.conf file

If the user wishes to permanently modify the value of a specific key, then the key-value pair must be stored within the file */etc/sysctl.conf*.

Step 1. In the Linux terminal, open the */etc/sysctl.conf* file using your favorite text editor. Run the following command on the Client1's terminal. When prompted for a password, type `password` and hit enter.

```
sudo featherpad /etc/sysctl.conf
```

This is a text file that can be edited in any text editor (`vim`, `nano`, etc.). For simplicity, we use a Graphical User Interface (GUI)-based text editor (`featherpad`).

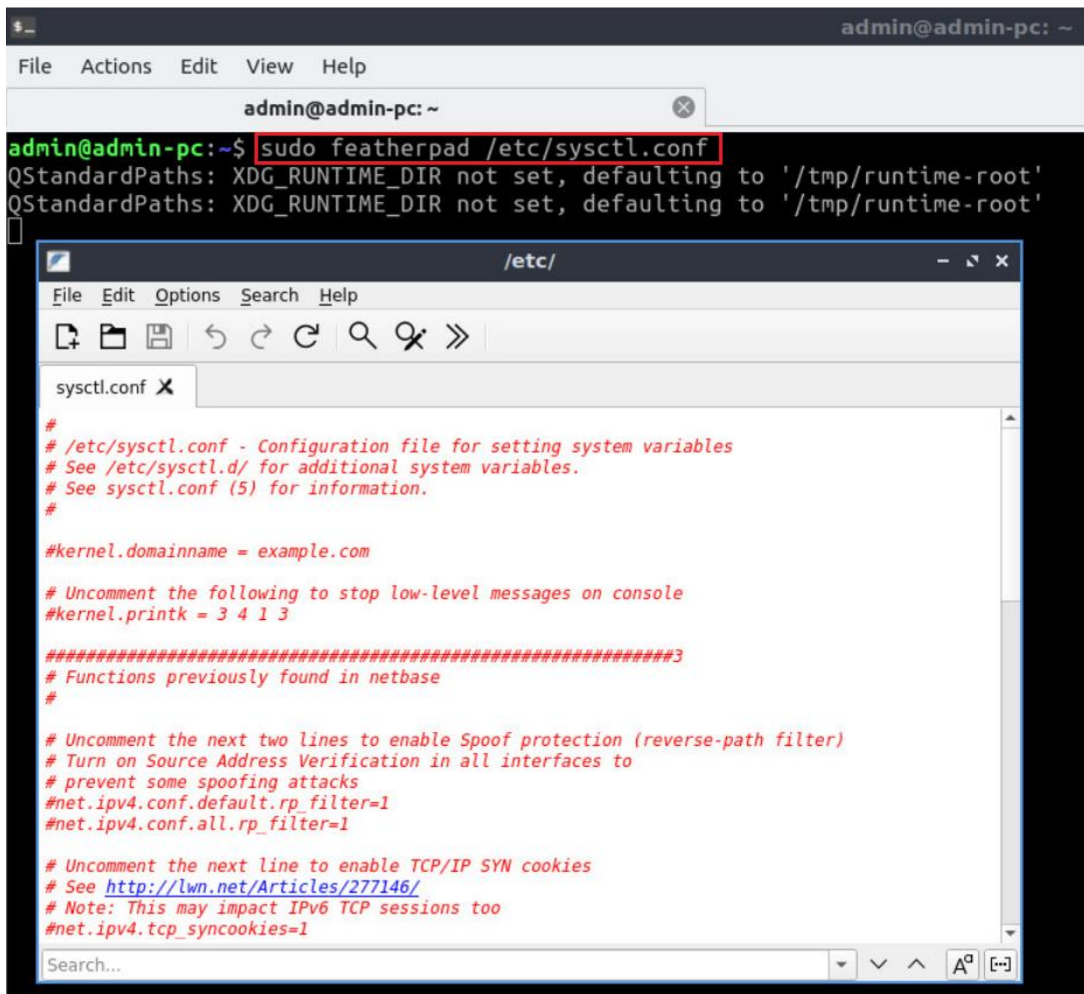

Figure 17. Opening the */etc/sysctl.conf* file.

Step 2. Keys and values are appended to this file. Enable IP forwarding permanently on the system by append `net.ipv4.ip_forward=1` to the */etc/sysctl.conf* file and save it. Once you have saved the file, close the text editor.

```
net.ipv4.ip_forward=1
```

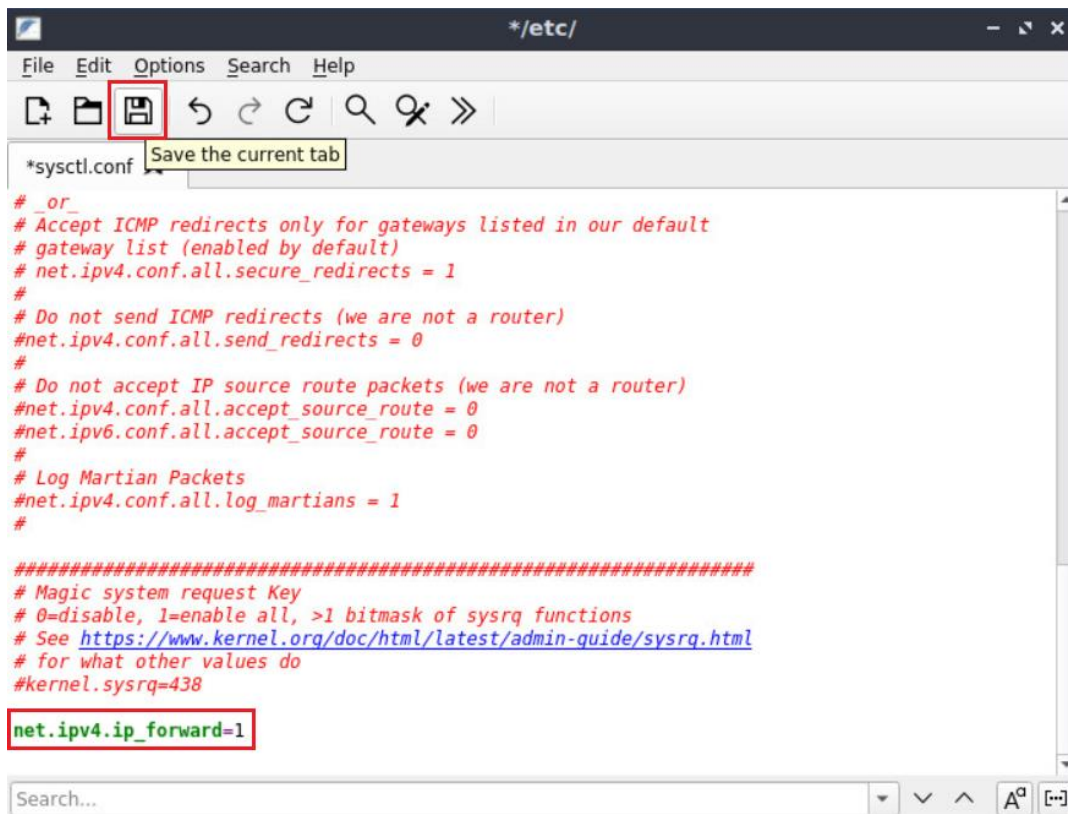

Figure 18. Appending key+value to the */etc/sysctl.conf* file and saving.

Step 3. To refresh the system with the new parameters, the `-p` switch is passed to the `sysctl` command as follows:

```
sudo sysctl -p
```

When prompted for a password, type `password` and hit enter.

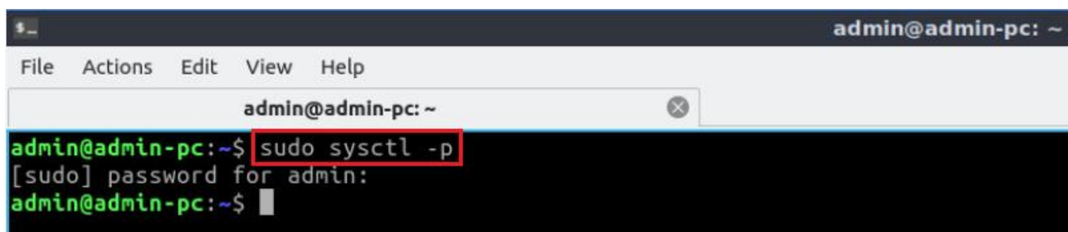

Figure 19. Loading new *sysctl.conf* parameters.

Now, even after a new system boot (or reboot), the system will have IP forwarding enabled.

4 Congestion control algorithms and sysctl

Congestion control algorithms can be inspected and modified using the `sysctl` command and the */etc/sysctl.conf* file. Specifically, the following operations are possible:

1. Check the installed congestion control algorithms on the system.

2. Inspect the default congestion control algorithm (i.e., the current algorithm used by the system).
3. Modify the congestion control algorithm.

4.1 Inspect and install/load congestion control algorithms

In Linux, it is possible to check the available TCP congestion control algorithms installed on the system with the command below.

Step 1. Execute the command below on the Client1's terminal.

```
sysctl net.ipv4.tcp_available_congestion_control
```

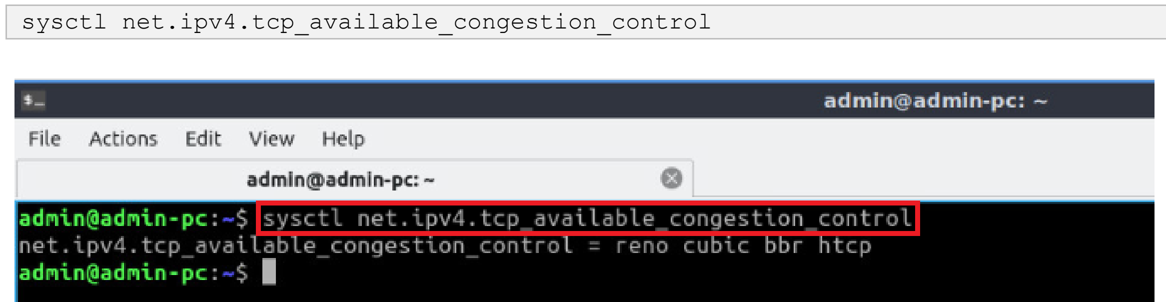

Figure 20. Displaying the system's available congestion control algorithms.

Usually, the default congestion control algorithm is CUBIC or Reno, depending on the installed operating system. A list of some of the possible output is:

- `reno`: Traditional TCP used by almost all other Operating Systems. Characterized by slow start, congestion avoidance, and fast retransmission via triple duplicate ACKs.
- `cubic`: CUBIC-TCP. Optimized congestion control algorithm for high bandwidth networks with high latency. Operates in a similar but more systematic fashion than BIC-TCP, in which its congestion window is a cubic function of time since the last packet loss, with the inflection point set to the window prior to the congestion event.
- `bic`: BIC-TCP. Congestion window utilizes a binary search algorithm to find the largest congestion window that will last the maximum amount of time.
- `htcp`: Hamilton TCP. A loss-based algorithm using additive-increase and multiplicative-decrease to control TCP's congestion window.
- `vegas`: TCP Vegas. Emphasizes packet delay, rather than packet loss, as a signal to help determine the rate at which to send packets.
- `bbr`: a new algorithm, discussed in future labs. Measures bottleneck bandwidth and Round-Trip Propagation (RTP) time in its execution of congestion control.

If the above command does not return a specific congestion control algorithm, it means that it is not loaded on the distribution.

Step 2. The command used in Step 1 listed three algorithms: `reno cubic bbr`. To install a new algorithm, its corresponding kernel module must be loaded. This can be done using

`insmod` or `modprobe` commands. For example, to load the BIC-TCP module, use the following command on the Client1's terminal:

```
sudo modprobe tcp_bic
```

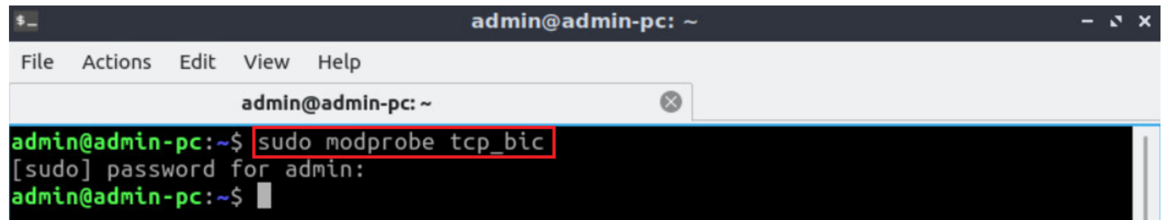

Figure 21. Loading `tcp_bic` module into the Linux kernel.

`modprobe` and `insmod` commands require high `sudo` privileges to insert kernel modules. When prompted for a password, type `password` and hit enter.

Step 3. To verify that the BIC-TCP algorithm is loaded, execute the below command on the Client1's terminal.

```
sysctl net.ipv4.tcp_available_congestion_control
```

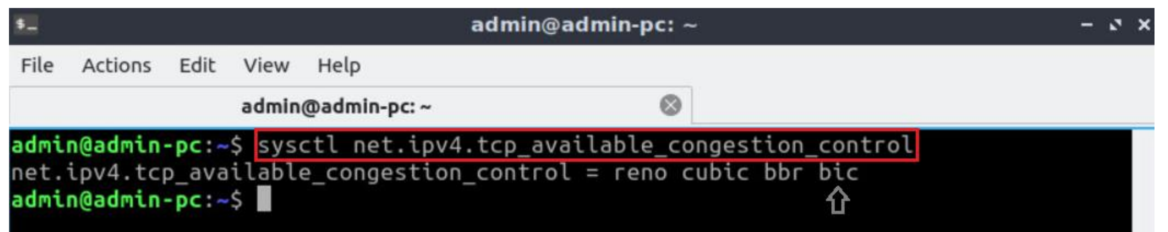

Figure 22. Displaying the system's available congestion control algorithms after loading TCP-BIC.

4.2 Inspect the default (current) congestion control algorithm

To check which TCP congestion control is currently being used by the Linux kernel, the *net.ipv4.tcp_congestion_control sysctl* key is read. This key can be read on an end-host's terminal (host h1 or host h2) or on the Client1's terminal.

Step 1. Execute the following command on the Client1's terminal to determine the current congestion control algorithm.

```
sysctl net.ipv4.tcp_congestion_control
```

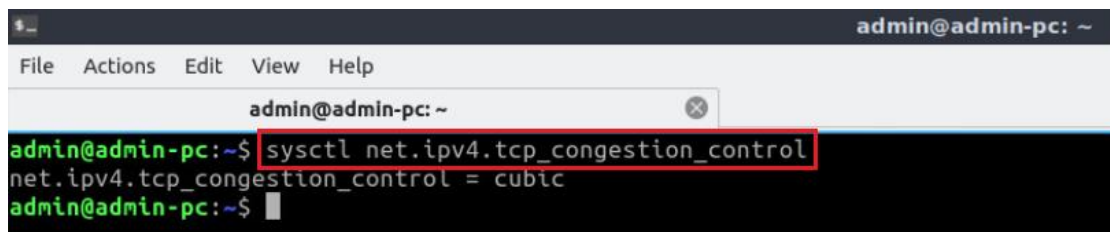

Figure 23. Current TCP congestion control algorithm.

The output shows that the default congestion control algorithm is Cubic. Note that applications can set this value (congestion control algorithm) for individual connections.

4.3 Modify the default (current) congestion control algorithm

To temporarily change the TCP congestion control algorithm, the `sysctl` command is used with the `-w` switch on the *net.ipv4.tcp_congestion_control* key.

Step 1. To modify the current algorithm to TCP Reno, the following command is used. Execute the command below on the Client1's terminal. When prompted for a password, type `password` and hit enter.

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

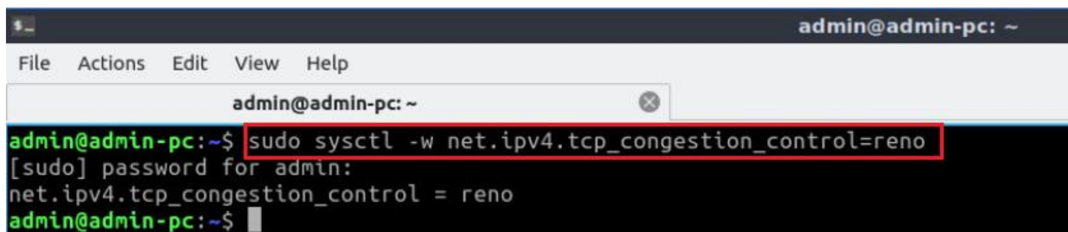

Figure 24. Modifying the congestion control algorithm to `reno`.

If no error occurred in the assignment (e.g., the module is not installed on the system), the output echoes back the new key-value pair, i.e.:
`net.ipv4.tcp_congestion_control=reno`

Step 2. Execute the following command on the Client1's terminal to determine the current congestion control algorithm.

```
sysctl net.ipv4.tcp_congestion_control
```

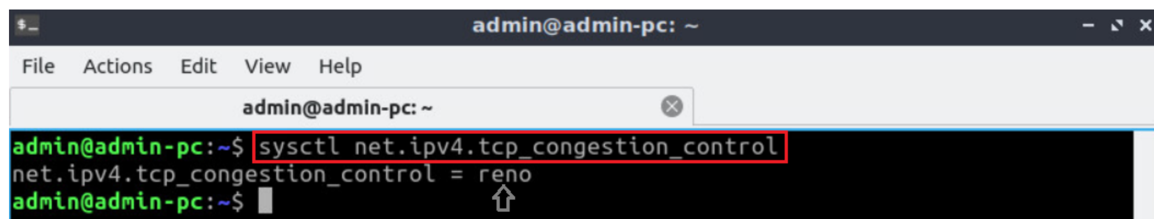

Figure 25. Current TCP congestion control algorithm after modifying to `reno`.

The output shows that the default congestion control algorithm is now Reno instead of Cubic.

5 iPerf3 throughput test

In this section, the throughput between host h1 and host h2 is measured using different congestion control algorithms, namely Reno, HTCP, and Cubic. Moreover, the test is

repeated using various injected delays to observe the throughput variations depending on each congestion control algorithm and the selected RTT.

5.1 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.

5.1.1 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
```

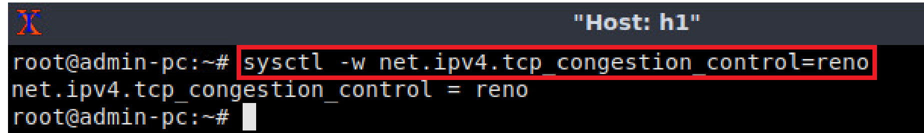

Figure 26. Changing TCP congestion control algorithm to `reno` on host h1.

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

```
iperf3 -s
```

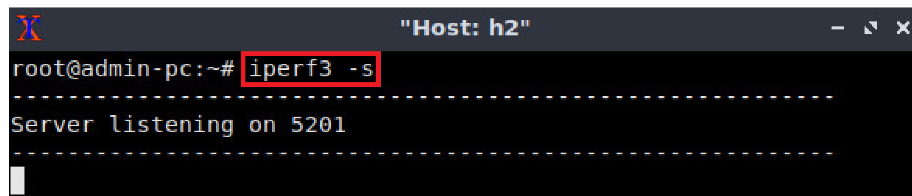

Figure 27. Starting iPerf3 server on host h2.

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. Note that this option is a capitalized 'O', not a zero.

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

```
"Host: h1"
root@admin-pc:~# iperf3 -c 10.0.0.2 -t 20 -O 10
Connecting to host 10.0.0.2, port 5201
[ 15] local 10.0.0.1 port 34490 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[ 15]   0.00-1.00   sec  1.13 GBytes  9.69 Gbits/sec  315   1.29 MBytes       (omitted)
[ 15]   1.00-2.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.72 MBytes       (omitted)
[ 15]   2.00-3.00   sec  1.11 GBytes  9.57 Gbits/sec    0   2.47 MBytes       (omitted)
[ 15]   3.00-4.00   sec  1.11 GBytes  9.56 Gbits/sec  135    703 KBytes       (omitted)
[ 15]   4.00-5.00   sec  1.11 GBytes  9.56 Gbits/sec    0   1.90 MBytes       (omitted)
[ 15]   5.00-6.00   sec  1.11 GBytes  9.57 Gbits/sec   90   1.15 MBytes       (omitted)
[ 15]   6.00-7.00   sec  1.11 GBytes  9.56 Gbits/sec    0   2.12 MBytes       (omitted)
[ 15]   7.00-8.00   sec  1.11 GBytes  9.57 Gbits/sec  180   1.20 MBytes       (omitted)
[ 15]   8.00-9.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.11 MBytes       (omitted)
[ 15]   1.00-1.00   sec  1.11 GBytes  4.79 Gbits/sec   45   1.83 MBytes
[ 15]   1.00-2.00   sec  1.11 GBytes  9.56 Gbits/sec   90   1.27 MBytes
[ 15]   2.00-3.00   sec  1.11 GBytes  9.56 Gbits/sec  180    730 KBytes
[ 15]   3.00-4.00   sec  1.11 GBytes  9.56 Gbits/sec   90    782 KBytes
[ 15]   4.00-5.00   sec  1.11 GBytes  9.55 Gbits/sec    0   1.93 MBytes
[ 15]   5.00-6.00   sec  1.11 GBytes  9.57 Gbits/sec  225    824 KBytes
[ 15]   6.00-7.00   sec  1.11 GBytes  9.56 Gbits/sec   90    735 KBytes
[ 15]   7.00-8.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.65 MBytes
[ 15]   8.00-9.00   sec  1.11 GBytes  9.56 Gbits/sec  180    724 KBytes
[ 15]   9.00-10.00  sec  1.11 GBytes  9.57 Gbits/sec  135   1.08 MBytes
[ 15]  10.00-11.00  sec  1.11 GBytes  9.56 Gbits/sec  135    878 KBytes
[ 15]  11.00-12.00  sec  1.11 GBytes  9.56 Gbits/sec  225    321 KBytes
[ 15]  12.00-13.00  sec  1.11 GBytes  9.56 Gbits/sec    0   1.80 MBytes
[ 15]  13.00-14.00  sec  1.11 GBytes  9.56 Gbits/sec    0   2.53 MBytes
[ 15]  14.00-15.00  sec  1.11 GBytes  9.56 Gbits/sec    0   3.09 MBytes
[ 15]  15.00-16.00  sec  1.11 GBytes  9.57 Gbits/sec  180   1004 KBytes
[ 15]  16.00-17.00  sec  1.11 GBytes  9.56 Gbits/sec   90   1022 KBytes
[ 15]  17.00-18.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.09 MBytes
[ 15]  18.00-19.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.49 MBytes
[ 15]  19.00-20.00  sec  1.11 GBytes  9.56 Gbits/sec   90   1014 KBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[ 15]   0.00-20.00  sec  22.3 GBytes  9.56 Gbits/sec  1890             sender
[ 15]   0.00-20.04  sec  22.3 GBytes  9.56 Gbits/sec                  receiver

iperf Done.
root@admin-pc:~#
```

Figure 28. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughput is 9.56 Gbps (sender) and 9.56 Gbps (receiver), and the number of retransmissions is 1890 (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.

5.1.2 Hamilton TCP (HTCP)

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

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

```
"Host: h1"
root@admin-pc:~# sysctl -w net.ipv4.tcp_congestion_control=htcp
net.ipv4.tcp_congestion_control = htcp
root@admin-pc:~#
```

Figure 29. Changing TCP congestion control algorithm to `htcp` on host h1.

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

```
iperf3 -s
```

```
"Host: h2"
root@admin-pc:~# iperf3 -s
-----------------------------------------------------------
Server listening on 5201
-----------------------------------------------------------
```

Figure 30. 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
```

```
"Host: h1"
root@admin-pc:~# iperf3 -c 10.0.0.2 -t 20 -O 10
Connecting to host 10.0.0.2, port 5201
[ 15] local 10.0.0.1 port 34494 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[ 15]   0.00-1.00   sec  1.13 GBytes  9.69 Gbits/sec  158   4.16 MBytes       (omitted)
[ 15]   1.00-2.00   sec  1.11 GBytes  9.57 Gbits/sec   45   2.49 MBytes       (omitted)
[ 15]   2.00-3.00   sec  1.11 GBytes  9.56 Gbits/sec   90   1.45 MBytes       (omitted)
[ 15]   3.00-4.00   sec  1.11 GBytes  9.56 Gbits/sec  225    956 KBytes       (omitted)
[ 15]   4.00-5.00   sec  1.11 GBytes  9.57 Gbits/sec  135    713 KBytes       (omitted)
[ 15]   5.00-6.00   sec  1.11 GBytes  9.56 Gbits/sec    0   1.85 MBytes       (omitted)
[ 15]   6.00-7.00   sec  1.11 GBytes  9.56 Gbits/sec    0   2.54 MBytes       (omitted)
[ 15]   7.00-8.00   sec  1.11 GBytes  9.57 Gbits/sec   90   1.27 MBytes       (omitted)
[ 15]   8.00-9.00   sec  1.11 GBytes  9.56 Gbits/sec   90   1.44 MBytes       (omitted)
[ 15]   9.00-10.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.68 MBytes       (omitted)
[ 15]   0.00-1.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.38 MBytes
[ 15]   1.00-2.00   sec  1.11 GBytes  9.56 Gbits/sec   90   1.61 MBytes
[ 15]   2.00-3.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.43 MBytes
[ 15]   3.00-4.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.40 MBytes
[ 15]   4.00-5.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.77 MBytes
[ 15]   5.00-6.00   sec  1.11 GBytes  9.56 Gbits/sec  135    781 KBytes
[ 15]   6.00-7.00   sec  1.11 GBytes  9.56 Gbits/sec  135   1.51 MBytes
[ 15]   7.00-8.00   sec  1.11 GBytes  9.56 Gbits/sec    0   2.30 MBytes
[ 15]   8.00-9.00   sec  1.11 GBytes  9.57 Gbits/sec    0   2.89 MBytes
[ 15]   9.00-10.00  sec  1.11 GBytes  9.56 Gbits/sec  135   1.14 MBytes
[ 15]  10.00-11.00  sec  1.11 GBytes  9.56 Gbits/sec   90   1.03 MBytes
[ 15]  11.00-12.00  sec  1.11 GBytes  9.56 Gbits/sec  214    696 KBytes
[ 15]  12.00-13.00  sec  1.11 GBytes  9.55 Gbits/sec  135   1.26 MBytes
[ 15]  13.00-14.00  sec  1.11 GBytes  9.56 Gbits/sec  270    621 KBytes
[ 15]  14.00-15.00  sec  1.11 GBytes  9.56 Gbits/sec    0   1.81 MBytes
[ 15]  15.00-16.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.90 MBytes
[ 15]  16.00-17.00  sec  1.11 GBytes  9.56 Gbits/sec  225    622 KBytes
[ 15]  17.00-18.00  sec  1.11 GBytes  9.56 Gbits/sec    0   1.81 MBytes
[ 15]  18.00-19.00  sec  1.11 GBytes  9.56 Gbits/sec   90   1.14 MBytes
[ 15]  19.00-20.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.51 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[ 15]   0.00-20.00  sec  22.3 GBytes  9.56 Gbits/sec  1789             sender
[ 15]   0.00-20.04  sec  22.3 GBytes  9.56 Gbits/sec                  receiver

iperf Done.
root@admin-pc:~#
```

Figure 31. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughput is 9.56 Gbps (sender) and 9.56 Gbps (receiver), and the number of retransmissions is 1789 (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.

5.1.3 TCP Cubic

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

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

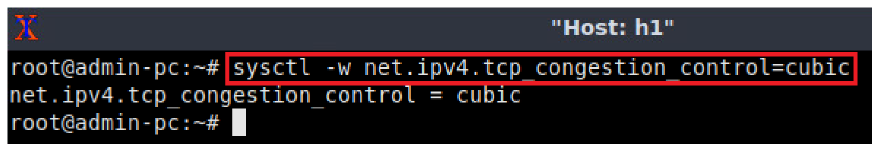

Figure 32. Changing TCP congestion control algorithm to `cubic` on host h1.

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

```
iperf3 -s
```

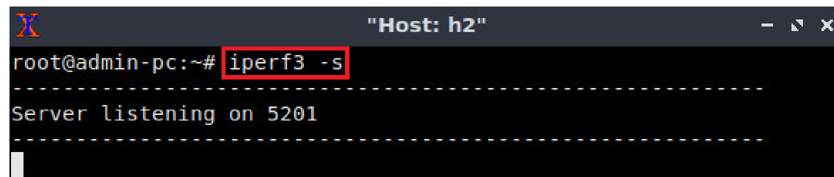

Figure 33. Starting iPerf3 server on host h2.

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

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

```
"Host: h1"
root@admin-pc:~# iperf3 -c 10.0.0.2 -t 20 -O 10
Connecting to host 10.0.0.2, port 5201
[ 15] local 10.0.0.1 port 34498 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[ 15]   0.00-1.00   sec  1.13 GBytes  9.69 Gbits/sec  135   5.90 MBytes       (omitted)
[ 15]   1.00-2.00   sec  1.11 GBytes  9.56 Gbits/sec   45   4.42 MBytes       (omitted)
[ 15]   2.00-3.00   sec  1.11 GBytes  9.56 Gbits/sec  135   1.76 MBytes       (omitted)
[ 15]   3.00-4.00   sec  1.11 GBytes  9.56 Gbits/sec  180   1.15 MBytes       (omitted)
[ 15]   4.00-5.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.43 MBytes       (omitted)
[ 15]   5.00-6.00   sec  1.11 GBytes  9.56 Gbits/sec  135    776 KBytes       (omitted)
[ 15]   6.00-7.00   sec  1.11 GBytes  9.56 Gbits/sec    0   1.48 MBytes       (omitted)
[ 15]   7.00-8.00   sec  1.11 GBytes  9.56 Gbits/sec  135   1.08 MBytes       (omitted)
[ 15]   8.00-9.00   sec  1.11 GBytes  9.57 Gbits/sec   90   1024 KBytes       (omitted)
[ 15]   1.00-1.00   sec  1.11 GBytes  4.78 Gbits/sec    0   1.84 MBytes
[ 15]   1.00-2.00   sec  1.11 GBytes  9.56 Gbits/sec  180   1.07 MBytes
[ 15]   2.00-3.00   sec  1.11 GBytes  9.56 Gbits/sec  135    970 KBytes
[ 15]   3.00-4.00   sec  1.11 GBytes  9.57 Gbits/sec  135   1.05 MBytes
[ 15]   4.00-5.00   sec  1.11 GBytes  9.56 Gbits/sec  180   1012 KBytes
[ 15]   5.00-6.00   sec  1.11 GBytes  9.56 Gbits/sec   45   1.25 MBytes
[ 15]   6.00-7.00   sec  1.11 GBytes  9.57 Gbits/sec   90   1.13 MBytes
[ 15]   7.00-8.00   sec  1.11 GBytes  9.56 Gbits/sec  135   1.22 MBytes
[ 15]   8.00-9.00   sec  1.11 GBytes  9.56 Gbits/sec  180    962 KBytes
[ 15]   9.00-10.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.15 MBytes
[ 15]  10.00-11.00  sec  1.11 GBytes  9.57 Gbits/sec   90   1.06 MBytes
[ 15]  11.00-12.00  sec  1.11 GBytes  9.56 Gbits/sec   90   1.22 MBytes
[ 15]  12.00-13.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.40 MBytes
[ 15]  13.00-14.00  sec  1.11 GBytes  9.56 Gbits/sec  135   1.08 MBytes
[ 15]  14.00-15.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.30 MBytes
[ 15]  15.00-16.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.46 MBytes
[ 15]  16.00-17.00  sec  1.11 GBytes  9.56 Gbits/sec   90   1.17 MBytes
[ 15]  17.00-18.00  sec  1.11 GBytes  9.56 Gbits/sec  135    984 KBytes
[ 15]  18.00-19.00  sec  1.11 GBytes  9.56 Gbits/sec   45   1.33 MBytes
[ 15]  19.00-20.00  sec  1.11 GBytes  9.56 Gbits/sec    0   1.87 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[ 15]   0.00-20.00  sec  22.3 GBytes  9.56 Gbits/sec  1845             sender
[ 15]   0.00-20.04  sec  22.3 GBytes  9.56 Gbits/sec                  receiver

iperf Done.
root@admin-pc:~#
```

Figure 34. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughput is 9.56 Gbps (sender) and 9.56 Gbps (receiver), and the number of retransmissions is 1845 (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.

5.2 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. On the client's terminal, run the following command to modify the previous rule to include 30ms delay. When prompted for a password, type `password` and hit enter.

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

Figure 35. 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 36. 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'*.

Figure 37. Modifying the TCP buffer size on host h2.

5.2.1 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 38. Changing TCP congestion control algorithm to `reno` on host h1.

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

```
iperf3 -s
```

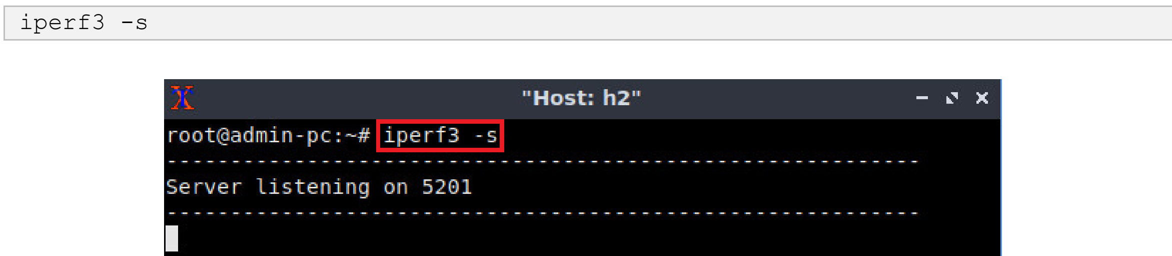

Figure 39. Starting iPerf3 server on host h2.

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
```

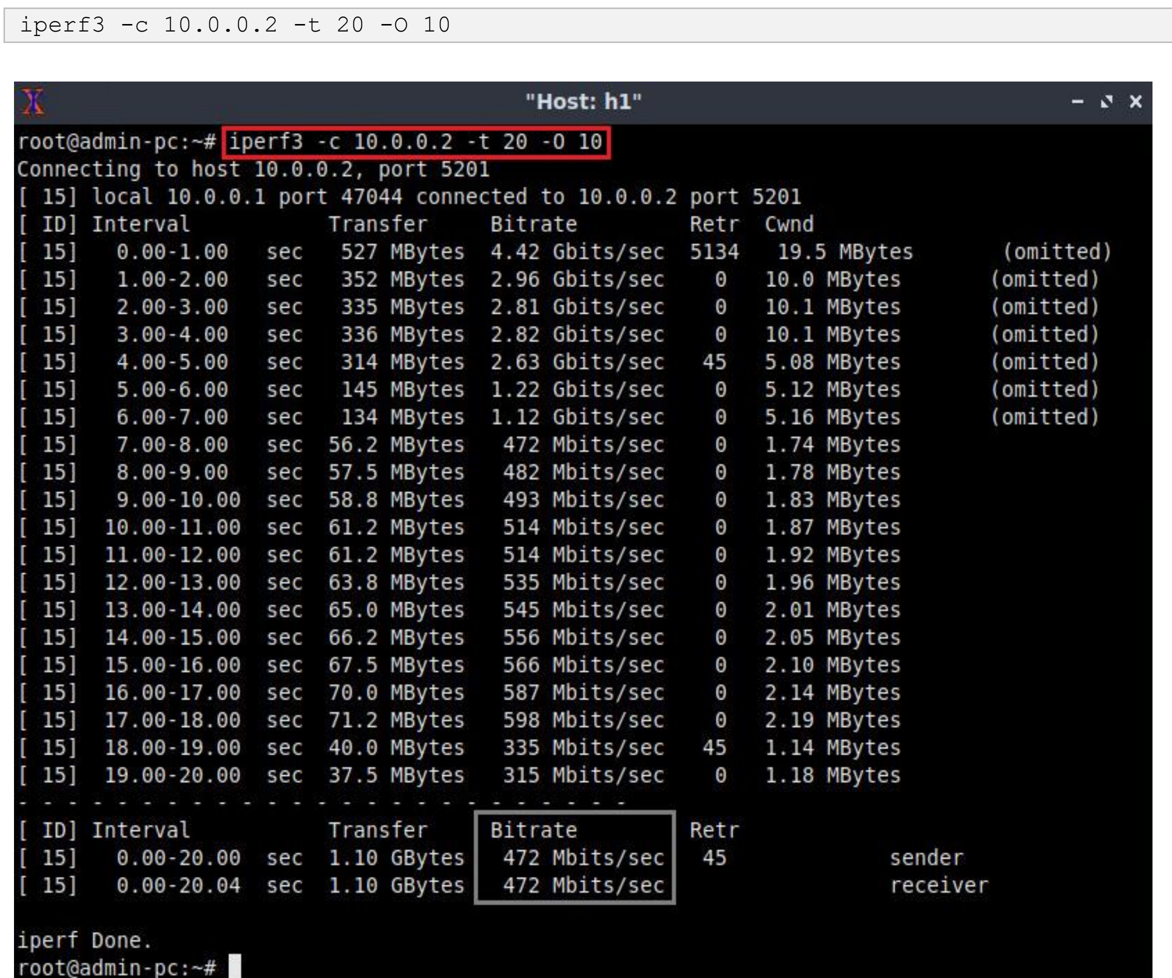

Figure 40. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughput is 472 Mbps (sender) and 472 Mbps (receiver), and the number of retransmissions is 45.

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.

5.2.2 Hamilton TCP (HTCP)

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

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

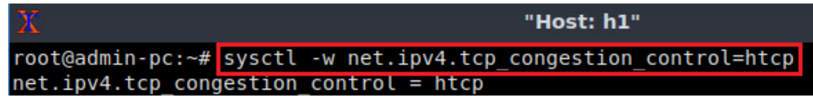

Figure 41. Changing TCP congestion control algorithm to `htcp` on host h1.

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

```
iperf3 -s
```

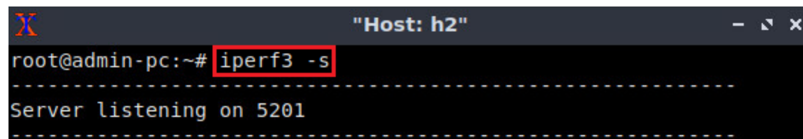

Figure 42. Starting iPerf3 server on host h2.

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

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

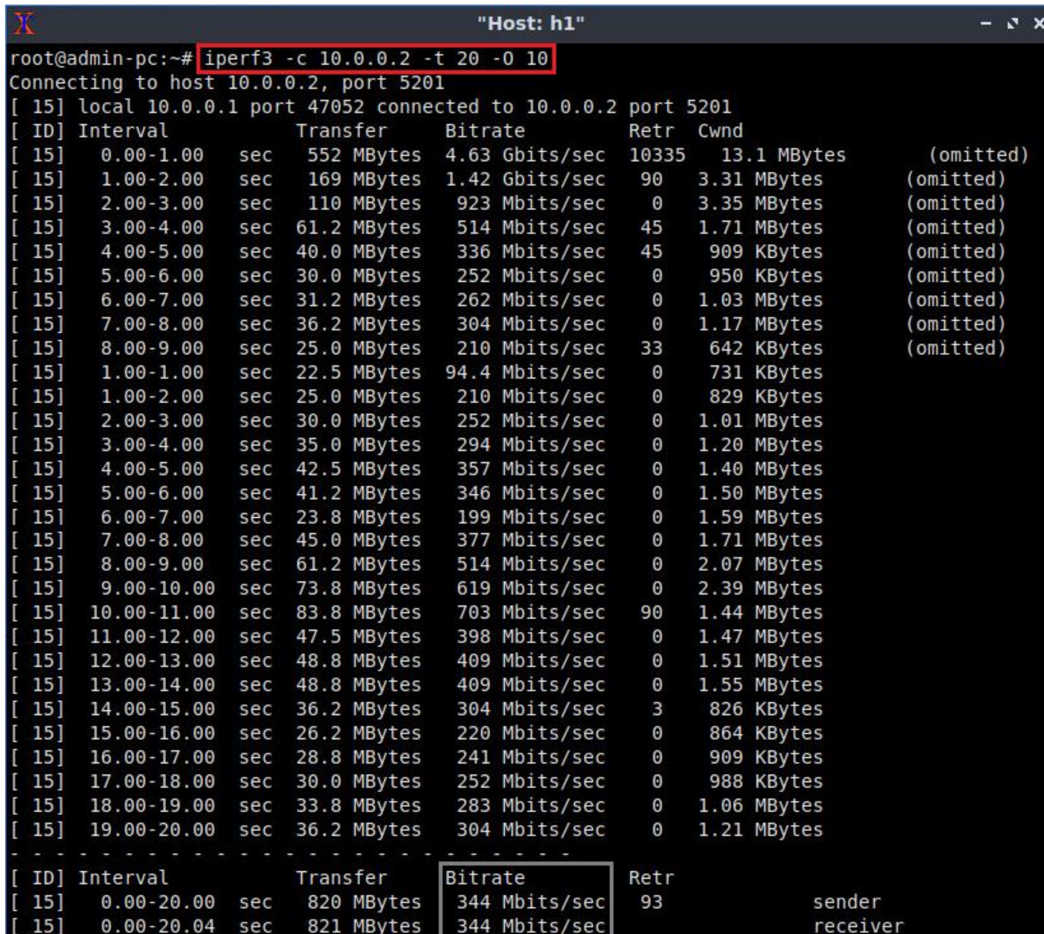

Figure 43. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughput is 344 Mbps (sender) and 344 Mbps (receiver), and the number of retransmissions is 93.

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.

5.2.3 TCP Cubic

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

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

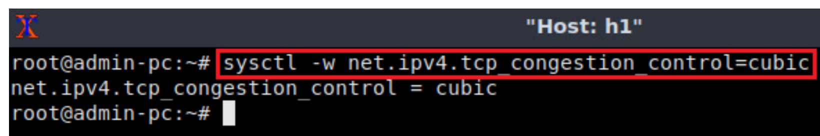

Figure 44. Changing TCP congestion control algorithm to `cubic` on host h1.

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

```
iperf3 -s
```

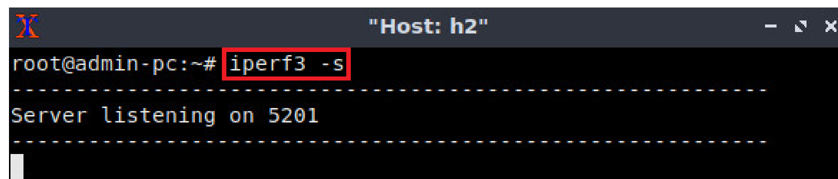

Figure 45. Starting iPerf3 server on host h2.

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

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

```
"Host: h1"
root@admin-pc:~# iperf3 -c 10.0.0.2 -t 20 -O 10
Connecting to host 10.0.0.2, port 5201
[ 15] local 10.0.0.1 port 47040 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate         Retr  Cwnd
[ 15]   0.00-1.00   sec   655 MBytes  5.49 Gbits/sec  24574   23.4 MBytes       (omitted)
[ 15]   1.00-2.00   sec   705 MBytes  5.91 Gbits/sec   45   16.9 MBytes       (omitted)
[ 15]   2.00-3.00   sec   564 MBytes  4.73 Gbits/sec    0   17.4 MBytes       (omitted)
[ 15]   3.00-4.00   sec   450 MBytes  3.78 Gbits/sec   45   12.6 MBytes       (omitted)
[ 15]   4.00-5.00   sec   348 MBytes  2.92 Gbits/sec   45   9.13 MBytes       (omitted)
[ 15]   5.00-6.00   sec   296 MBytes  2.49 Gbits/sec   45   6.63 MBytes       (omitted)
[ 15]   6.00-7.00   sec   224 MBytes  1.88 Gbits/sec    0   6.91 MBytes       (omitted)
[ 15]   7.00-8.00   sec   229 MBytes  1.92 Gbits/sec    0   7.15 MBytes       (omitted)
[ 15]   8.00-9.00   sec   176 MBytes  1.48 Gbits/sec   45   5.24 MBytes       (omitted)
[ 15]   1.00-1.00   sec   182 MBytes   765 Mbits/sec    0   5.61 MBytes
[ 15]   1.00-2.00   sec   172 MBytes  1.45 Gbits/sec   45   4.05 MBytes
[ 15]   2.00-3.00   sec   136 MBytes  1.14 Gbits/sec    0   4.24 MBytes
[ 15]   3.00-4.00   sec   145 MBytes  1.22 Gbits/sec    0   4.40 MBytes
[ 15]   4.00-5.00   sec   146 MBytes  1.23 Gbits/sec    0   4.53 MBytes
[ 15]   5.00-6.00   sec   146 MBytes  1.23 Gbits/sec   45   3.25 MBytes
[ 15]   6.00-7.00   sec   110 MBytes   923 Mbits/sec    0   3.42 MBytes
[ 15]   7.00-8.00   sec   116 MBytes   975 Mbits/sec    0   3.57 MBytes
[ 15]   8.00-9.00   sec   119 MBytes   996 Mbits/sec    0   3.68 MBytes
[ 15]   9.00-10.00  sec   122 MBytes  1.03 Gbits/sec    0   3.76 MBytes
[ 15]  10.00-11.00  sec   125 MBytes  1.05 Gbits/sec    0   3.83 MBytes
[ 15]  11.00-12.00  sec  96.2 MBytes   807 Mbits/sec   45   2.82 MBytes
[ 15]  10.00-11.00  sec   125 MBytes  1.05 Gbits/sec    0   3.83 MBytes
[ 15]  11.00-12.00  sec  96.2 MBytes   807 Mbits/sec   45   2.82 MBytes
[ 15]  12.00-13.00  sec  82.5 MBytes   692 Mbits/sec   45   2.08 MBytes
[ 15]  13.00-14.00  sec  70.0 MBytes   587 Mbits/sec    0   2.19 MBytes
[ 15]  14.00-15.00  sec  72.5 MBytes   608 Mbits/sec    0   2.28 MBytes
[ 15]  15.00-16.00  sec  76.2 MBytes   640 Mbits/sec    0   2.35 MBytes
[ 15]  16.00-17.00  sec  77.5 MBytes   650 Mbits/sec    0   2.40 MBytes
[ 15]  17.00-18.00  sec  80.0 MBytes   671 Mbits/sec    0   2.43 MBytes
[ 15]  18.00-19.00  sec  80.0 MBytes   671 Mbits/sec    0   2.45 MBytes
[ 15]  19.00-20.00  sec  81.2 MBytes   681 Mbits/sec    0   2.45 MBytes
- - - - - - - - - - - - - - - - - - - - - - - - -
[ ID] Interval           Transfer     Bitrate         Retr
[ 15]   0.00-20.00  sec  2.19 GBytes   938 Mbits/sec  180             sender
[ 15]   0.00-20.04  sec  2.19 GBytes   939 Mbits/sec                  receiver

iperf Done.
root@admin-pc:~#
```

Figure 46. Running iPerf3 client on host h1.

The figure above shows the iPerf3 test output report. The average achieved throughput is 938 Mbps (sender) and 939 Mbps (receiver), and the number of retransmissions is 180.

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.

This concludes Lab 6. Stop the emulation and then exit out of MiniEdit and Linux terminal.

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. 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.
3. 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.
4. 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.
5. 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.
6. 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.
7. System information variables – sysctl (7). [Online]. Available: https://www.kernel.org/doc/Documentation/networking/ip-sysctl.txt.