



UNIVERSITY OF
SOUTH CAROLINA

NETWORK TOOLS AND PROTOCOLS

Lab 8: Bandwidth-delay Product and TCP Buffer Size

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 buffers, BDP, and TCP window	3
1.1 TCP buffers	3
1.2 Bandwidth-delay product.....	4
1.3 Practical observations on setting TCP buffer size	5
1.4 TCP window size calculated value.....	7
1.5 Zero window.....	8
2 Lab topology.....	8
2.1 Starting host h1 and host h2	9
2.2 Emulating 10 Gbps high-latency WAN	10
3 BDP and buffer size	13
3.1 Window size in sysctl.....	13
4 Modifying buffer size and throughput test.....	15
References	17

Overview

This lab explains the bandwidth-delay product (BDP) and how to modify the TCP buffer size in Linux systems. Throughput measurements are also conducted to test and verify TCP buffer configurations based on the BDP.

Objectives

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

1. Understand BDP.
2. Define and calculate TCP window size.
3. Modify the TCP buffer size with `sysctl`, based on BDP calculations.
4. Emulate WAN properties in Mininet.
5. Achieve full throughput in WANs by modifying the size of TCP buffers.

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 buffers, BDP, and TCP window.
2. Section 2: Lab topology.
3. Section 3: BDP and buffer size experiments.
4. Section 4: Modifying buffer size and throughput test.

1 Introduction to TCP buffers, BDP, and TCP window

1.1 TCP buffers

The TCP send and receive buffers may impact the performance of Wide Area Networks (WAN) data transfers. Consider Figure 1. At the sender side, TCP receives data from the

application layer and places it in the TCP send buffer. Typically, TCP fragments the data in the buffer into maximum segment size (MSS) units. In this example, the MSS is 100 bytes. Each segment carries a sequence number, which is the byte-stream number of the first byte in the segment. The corresponding acknowledgement (Ack) carries the number of the next expected byte (e.g., Ack-101 acknowledges bytes 1-100, carried by the first segment). At the receiver, TCP receives data from the network layer and places it into the TCP receive buffer. TCP delivers the data *in order* to the application. E.g., bytes contained in a segment, say segment 2 (bytes 101-200), cannot be delivered to the application layer before the bytes contained in segment 1 (bytes 1-100) are delivered to the application. At any given time, the TCP receiver indicates the TCP sender how many bytes the latter can send, based on how much free buffer space is available at the receiver.

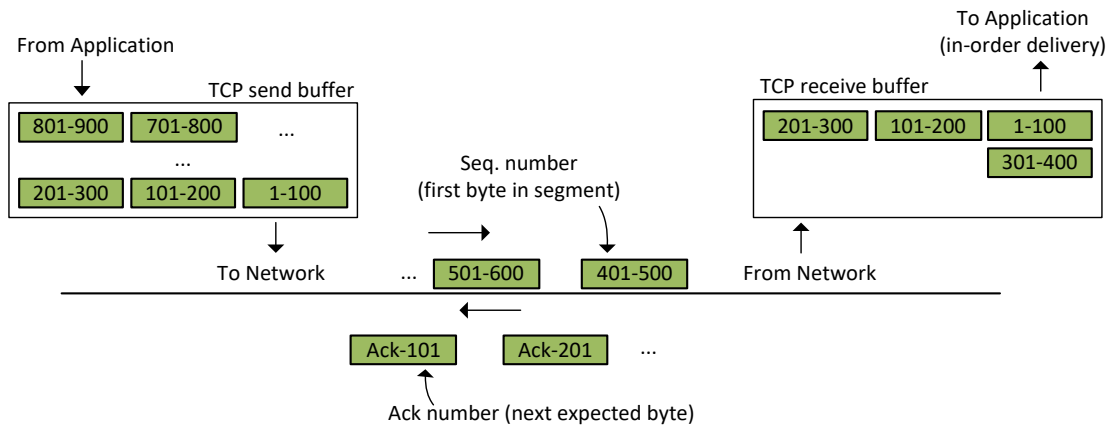


Figure 1. TCP send and receive buffers.

1.2 Bandwidth-delay product

In many WANs, the round-trip time (RTT) is dominated by the propagation delay. Long RTTs along and TCP buffer size have throughput implications. Consider a 10 Gbps WAN with a 50-millisecond RTT. Assume that the TCP send and receive buffer sizes are set to 1 Mbyte (1 Mbyte = 1024^2 bytes = 1,048,576 bytes = $1,048,576 \cdot 8$ bits = 8,388,608 bits). With a bandwidth (Bw) of 10 Gbps, this number of bits is approximately transmitted in

$$T_{tx} = \frac{\# \text{ bits}}{Bw} = \frac{8,388,608}{10 \cdot 10^9} = 0.84 \text{ milliseconds.}$$

I.e., after 0.84 milliseconds, the content of the TCP send buffer will be completely sent. At this point, TCP must wait for the corresponding acknowledgements, which will only start arriving at $t = 50$ milliseconds. This means that the sender only uses $0.84/50$ or 1.68% of the available bandwidth.

The solution to that above problem lies in allowing the sender to continuously transmit segments until the corresponding acknowledgments arrive back. Note that the first acknowledgement arrives after an RTT. The number of bits that can be transmitted in a RTT period is given by the bandwidth of the channel in bits per second multiplied by the

RTT. This quantity is referred to as the bandwidth-delay product (BDP). For the above example, the buffer size must be greater than or equal to the BDP:

$$\text{TCP buffer size} \geq \text{BDP} = (10 \cdot 10^9)(50 \cdot 10^{-3}) = 500,000,000 \text{ bits} = 62,500,000 \text{ bytes.}$$

The first factor ($10 \cdot 10^9$) is the bandwidth; the second factor ($50 \cdot 10^{-3}$) is the RTT. For practical purposes, the TCP buffer can be also expressed in Mbytes (1 Mbyte = 1024^2 bytes) or Gbytes (1 Gbyte = 1024^3 bytes). The above expression, in Mbytes, is

$$\text{TCP buffer size} \geq 62,500,000 \text{ bytes} = 59.6 \text{ Mbytes} \approx 60 \text{ Mbytes.}$$

1.3 Practical observations on setting TCP buffer size

Linux systems configuration. Linux assumes that half of the send/receive TCP buffers are used for internal structures. Thus, only half of the buffer size is used to store segments. This implies that if a TCP connection requires certain buffer size, then the administrator must configure the buffer size equals to twice that size. For the previous example, the TCP buffer size must be:

$$\text{TCP buffer size} \geq 2 \cdot 60 \text{ Mbytes} = 120 \text{ Mbytes.}$$

Packet loss scenarios and TCP BBR¹. TCP provides a reliable, in-order delivery service. Reliability means that bytes successfully received must be acknowledged. In-order delivery means that the receiver only delivers bytes to the application layer in sequential order. The memory occupied by those bytes will be deallocated from the receive buffer after passing the bytes to the application layer. This process has some performance implications, as illustrated next. Consider Figure 2, which shows a TCP receive buffer. Assume that the segment carrying bytes 101-200 is lost. Although the receiver has successfully received bytes 201-900, it cannot deliver to the application layer until bytes 101-200 are received. Note that the receive buffer may become full, which would block the sender from utilizing the channel.

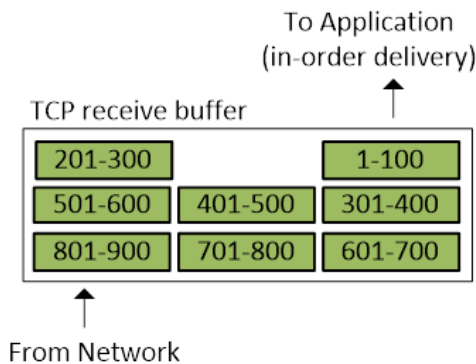


Figure 2. TCP receive buffer. Although bytes 301-900 have been received, they cannot be delivered to the application until the segment carrying bytes 201-300 are received.

While setting the buffer size equal to the BDP is acceptable when traditional congestion control algorithms are used (e.g., Reno², Cubic³, HTCP⁴), this size may not allow the full

utilization of the channel with BBR¹. In contrast to other algorithms, BBR does not reduce the transmission rate after a packet loss. For example, suppose that a packet sent at $t = 0$ is lost, as shown in Figure 3. At $t = \text{RTT}$, the acknowledgement identifying the packet to retransmit is received. By then, the sender has sent BDP bits, which will be stored in the receive buffer. This data cannot be delivered yet to the application, because of the in-order delivery requirement. Since the receive buffer has a capacity of BDP only, the sender is temporarily blocked until the acknowledgement for the retransmitted data is received at $t = 2 \cdot \text{RTT}$. Thus, the throughput over the period $t = 0$ to $t = 2 \cdot \text{RTT}$ is reduced by half:

$$\text{throughput} = \frac{\text{\# bits transmitted}}{\text{period}} = \frac{Bw \cdot \text{RTT}}{2 \cdot \text{RTT}} = \frac{Bw}{2}$$

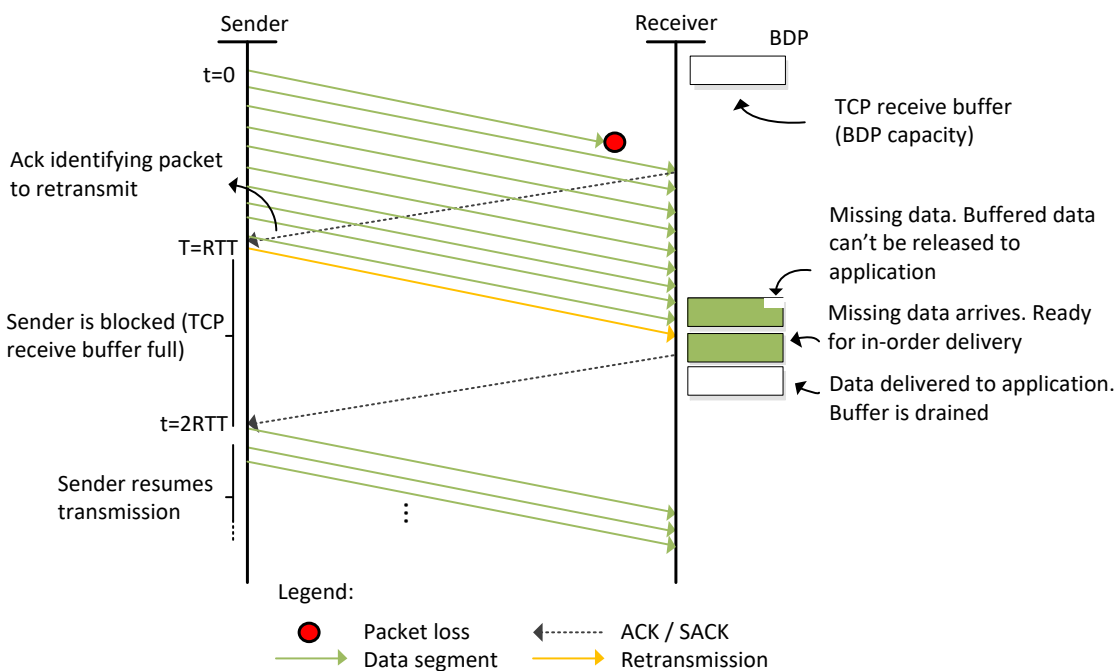


Figure 3. A scenario where a TCP receive buffer size of BDP cannot prevent throughput degradation.

With BBR, to fully utilize the available bandwidth, the TCP send and receive buffers must be large enough to prevent such situation. Figure 4 shows an example on how a TCP buffer size of $2 \cdot \text{BDP}$ mitigates packet loss.

High to moderate packet loss scenarios, using TCP BBR:
 TCP send/receive buffer \geq several BDPs (e.g., $4 \cdot \text{BDP}$)

Continuing with the example of Section 1.2, in a Linux system using TCP BBR, the send/receive buffers for a BDP of 60 Mbytes in a high to moderate packet loss scenario should be:

$$\text{TCP buffer size} \geq (2 \cdot 60 \text{ Mbytes}) \cdot 4 = 480 \text{ Mbytes.}$$

The factor 2 is because of the Linux systems configuration, and the factor 4 is because of the use of TCP BBR in a high to moderate packet loss scenario.

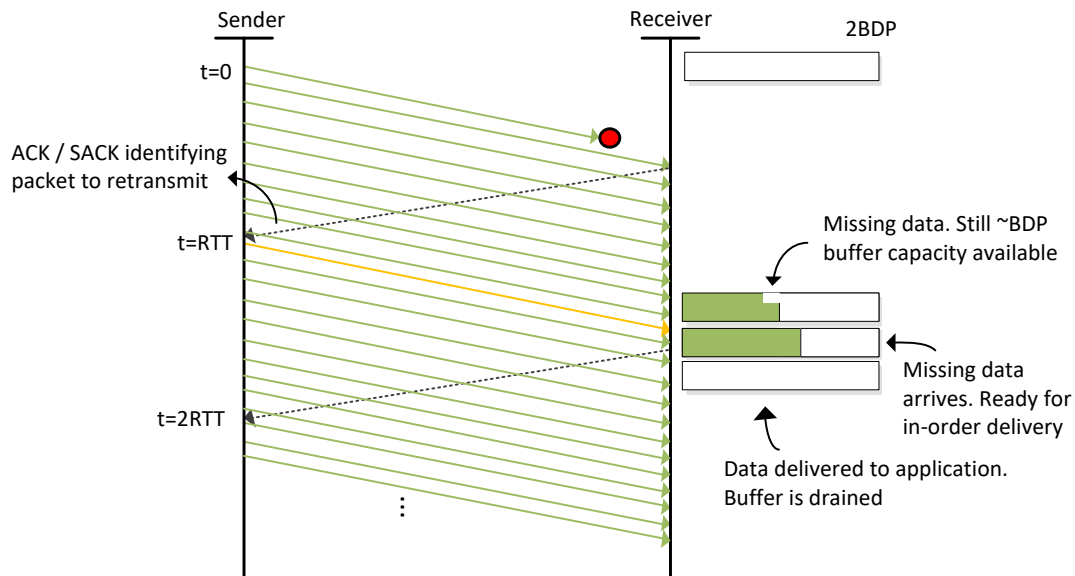


Figure 4. A scenario where a TCP buffer size of $2 \cdot \text{BDP}$ mitigates packet loss.

1.4 TCP window size calculated value

The receiver must constantly communicate with the sender to indicate how much free buffer space is available in the TCP receive buffer. This information is carried in a TCP header field called window size. The window size has a maximum value of 65,535 bytes, as the header value allocated for the window size is two bytes long (16 bits; $2^{16}-1 = 65,535$). However, this value is not large enough for high-bandwidth high-latency networks. Therefore, *TCP window scale option* was standardized in RFC 1323⁵. By using this option, the calculated window size may be increased up to a maximum value of 1,073,725,440 bytes. When advertising its window, a device also advertises the *scale factor* (multiplier) that will be used throughout the session. The TCP window size is calculated as follows:

$$\text{Scaled TCP}_{\text{Win}} = \text{TCP}_{\text{Win}} \cdot \text{Scaling Factor}$$

As an example, consider the following example. For an advertised TCP window of 2,049 and a scale factor of 512, then the final window size is 1,049,088 bytes. Figure 5 displays a packet inspected in Wireshark protocol analyzer for this numerical example.

```

▶ Flags: 0x010 (ACK)
  Window size value: 2049
  [Calculated window size: 1049088]
  [Window size scaling factor: 512]
    
```

Figure 5. Window Scaling in Wireshark.

1.5 Zero window

When the TCP buffer is full, a window size of zero is advertised to inform the other end that it cannot receive any more data. When a client sends a TCP window of zero, the server will pause its data transmission, and waits for the client to recover. Once the client is recovered, it digests data, and inform the server to resume the transmission again by setting again the TCP window.

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.

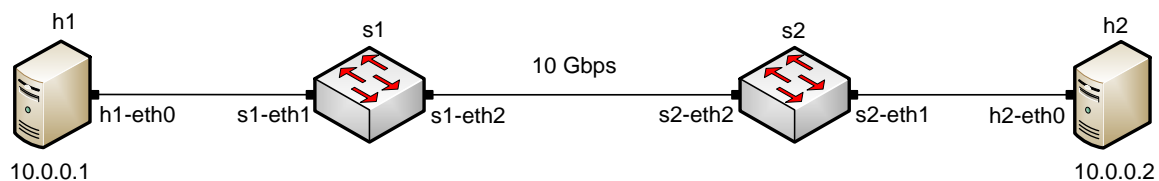


Figure 6. Lab topology.

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`.

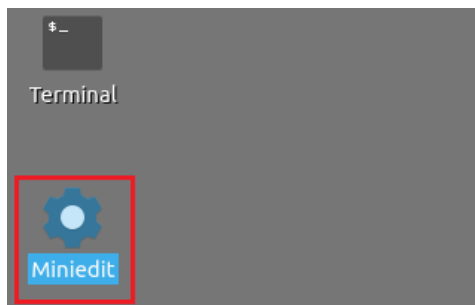


Figure 7. Miniedit shortcut.

Step 2. On Miniedit's menu bar, click on *File* then *Open* to load the lab's topology. Locate the `lab8.mn` topology file and click on *Open*.

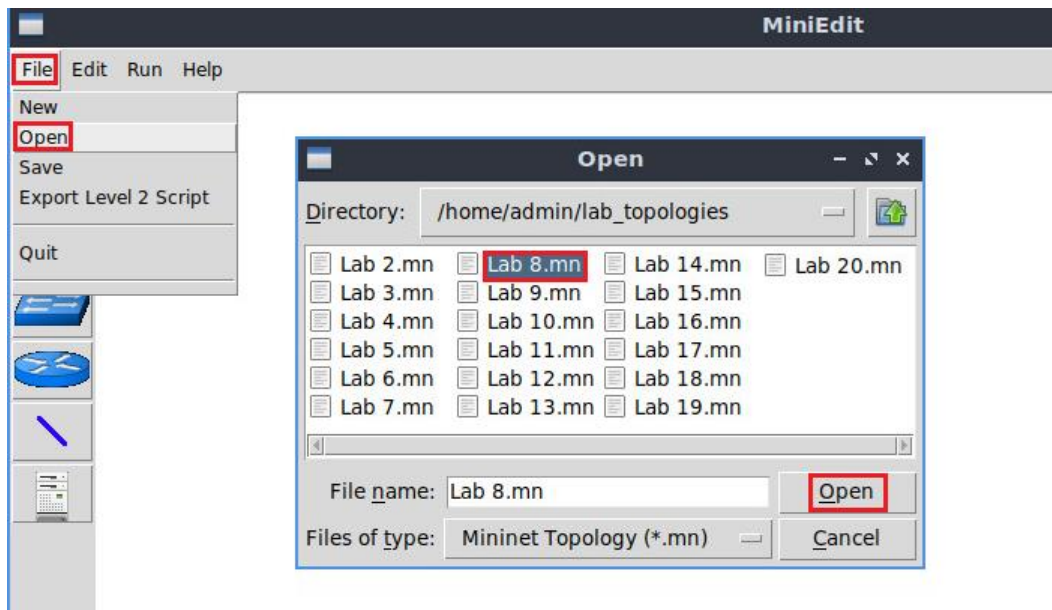


Figure 8. Miniedit's *Open* dialog.

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.

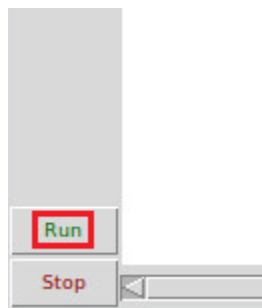


Figure 9. Running the emulation.

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.

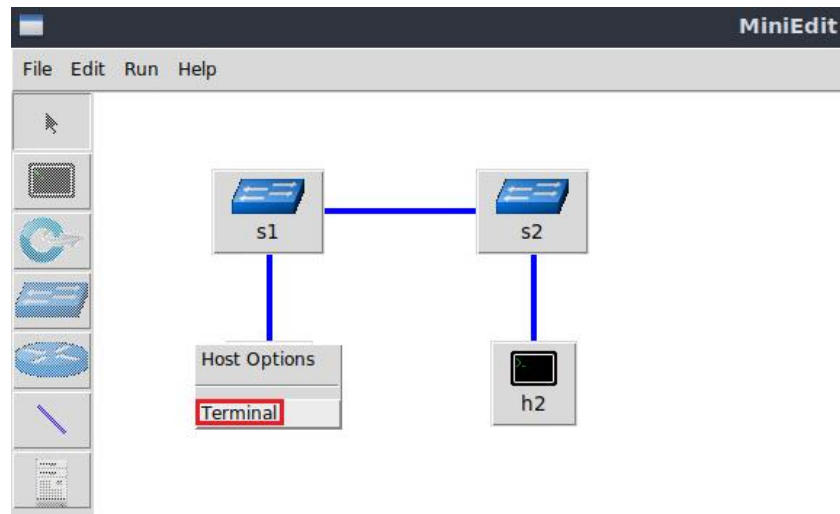


Figure 10. Opening a terminal on host h1.

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.

```

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 11. Connectivity test using `ping` command.

Figure 11 indicates that there is connectivity between host h1 and host h2.

2.2 Emulating 10 Gbps high-latency WAN

This section emulates a high-latency WAN by introducing delays to the network. We will first set the bandwidth between hosts 1 and 2 to 10 Gbps. Then, we will emulate a 20 ms delay 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.

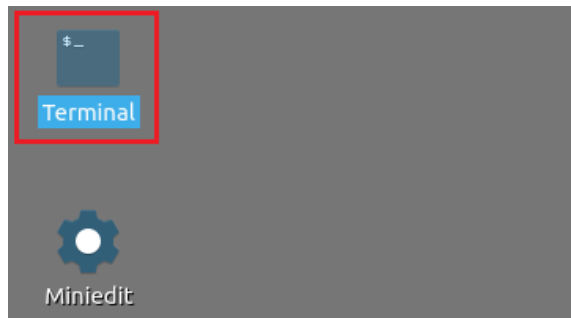
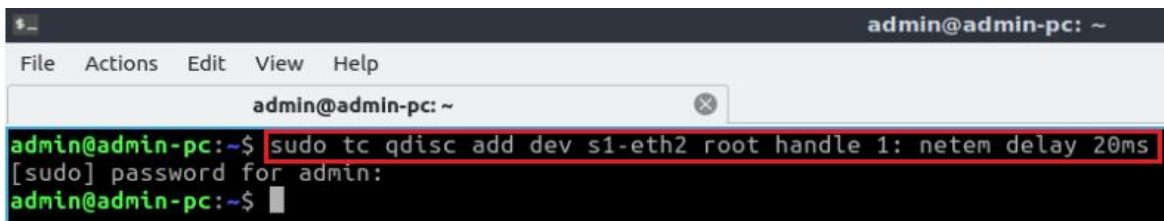


Figure 12. Shortcut to open a Linux terminal.

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. This command introduces 20ms delay on S1's `s1-eth2` interface.

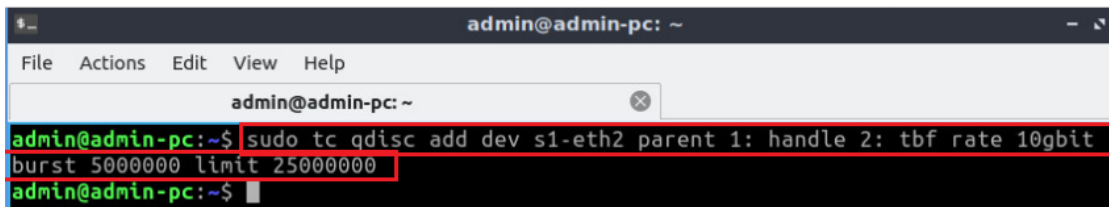
```
sudo tc qdisc add dev s1-eth2 root handle 1: netem delay 20ms
```

Figure 13. Adding 20ms delay to switch S1's `s1-eth2` interface.

Step 3. Modify the bandwidth of the link connecting the switches S1 and S2: on the same terminal, type the command below. This command sets the bandwidth to 10 Gbps on S1's `s1-eth2` interface. The `tbfb` parameters are the following:

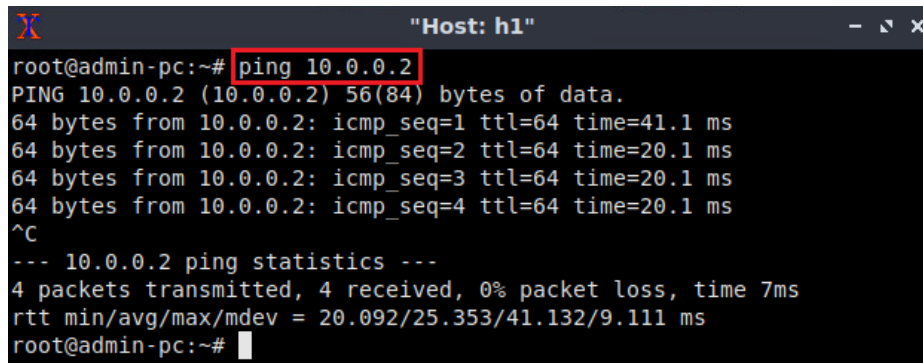
- `rate`: 10gbit
- `burst`: 5,000,000
- `limit`: 25,000,000

```
sudo tc qdisc add dev s1-eth2 parent 1: handle 2: tbf rate 10gbit burst 5000000
limit 25000000
```

Figure 14. Limiting the bandwidth to 10 Gbps on switch S1's `s1-eth2` interface.

Step 3. On h1's terminal, type `ping 10.0.0.2`. This command tests the connectivity between host h1 and host h2. The test was initiated by h1 as the command is executed on h1's terminal.

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=41.1 ms
64 bytes from 10.0.0.2: icmp_seq=2 ttl=64 time=20.1 ms
64 bytes from 10.0.0.2: icmp_seq=3 ttl=64 time=20.1 ms
64 bytes from 10.0.0.2: icmp_seq=4 ttl=64 time=20.1 ms
^C
--- 10.0.0.2 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 7ms
rtt min/avg/max/mdev = 20.092/25.353/41.132/9.111 ms
root@admin-pc:~#

```

Figure 15. 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 20.092, 25.353, 41.132, and 9.111 milliseconds, respectively. The output above verifies that delay was injected successfully, as the RTT is approximately 20ms.

Step 4. The user can now verify the rate limit configuration by using the `iperf3` tool to measure throughput. To launch iPerf3 in server mode, run the command `iperf3 -s` in H2's terminal:

```
iperf3 -s
```



```

root@admin-pc:~# iperf3 -s
-----
Server listening on 5201
-----

```

Figure 16. Host h2 running `iperf3` as server.

Step 5. Now to launch iPerf3 in client mode again by running the command `iperf3 -c 10.0.0.2` in h1's terminal:

```
iperf3 -c 10.0.0.2
```

```

Host: h1
root@admin-pc:~# iperf3 -c 10.0.0.2
Connecting to host 10.0.0.2, port 5201
[ 15] local 10.0.0.1 port 47090 connected to 10.0.0.2 port 5201
[ ID] Interval          Transfer          Bitrate          Retr  Cwnd
[ 15]  0.00-1.00      sec    321 MBytes    2.69 Gbits/sec    315  18.1 MBytes
[ 15]  1.00-2.00      sec    391 MBytes    3.28 Gbits/sec     0  18.1 MBytes
[ 15]  2.00-3.00      sec    321 MBytes    2.69 Gbits/sec     0  18.1 MBytes
[ 15]  3.00-4.00      sec    385 MBytes    3.23 Gbits/sec     0  18.1 MBytes
[ 15]  4.00-5.00      sec    389 MBytes    3.26 Gbits/sec     0  18.1 MBytes
[ 15]  5.00-6.00      sec    382 MBytes    3.21 Gbits/sec     0  18.1 MBytes
[ 15]  6.00-7.00      sec    388 MBytes    3.25 Gbits/sec     0  18.1 MBytes
[ 15]  7.00-8.00      sec    396 MBytes    3.32 Gbits/sec     0  18.1 MBytes
[ 15]  8.00-9.00      sec    396 MBytes    3.32 Gbits/sec     0  18.1 MBytes
[ 15]  9.00-10.00     sec    394 MBytes    3.30 Gbits/sec     0  18.1 MBytes
-----
[ ID] Interval          Transfer          Bitrate          Retr
[ 15]  0.00-10.00     sec    3.67 GBytes    3.16 Gbits/sec    315
[ 15]  0.00-10.04     sec    3.67 GBytes    3.14 Gbits/sec
iperf Done.
root@admin-pc:~#

```

Figure 17. iPerf3 throughput test.

Notice the measured throughput now is approximately 3 Gbps, which is different than the value assigned in our `tbw` rule. Next, we explain why the 10 Gbps maximum theoretical bandwidth is not achieved.

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 BDP and buffer size

In connections that have a small BDP (either because the link has a low bandwidth or because the latency is small), buffers are usually small. However, in high-bandwidth high-latency networks, where the BDP is large, a larger buffer is required to achieve the maximum theoretical bandwidth.

3.1 Window size in `sysctl`

The tool `sysctl` is used for dynamically changing parameters in the Linux operating system. It allows users to modify kernel parameters dynamically without rebuilding the Linux kernel.

The `sysctl` key for the receive window size is `net.ipv4.tcp_rmem` and the send window size is `net.ipv4.tcp_wmem`

Step 1. To read the current receiver window size value of host h1, use the following command on h1's terminal:

```
sysctl net.ipv4.tcp_rmem
```

```

X "Host: h1"
root@admin-pc:~# sysctl net.ipv4.tcp_rmem
net.ipv4.tcp_rmem = 10240      87380      16777216
root@admin-pc:~# █

```

Figure 18. Receive window read in `sysctl`.

Step 2. To read the current send window size value of host h1, use the following command on h1's terminal:

```
sysctl net.ipv4.tcp_wmem
```

```

X "Host: h1"
root@admin-pc:~# sysctl net.ipv4.tcp_wmem
net.ipv4.tcp_wmem = 10240      87380      16777216
root@admin-pc:~# █

```

Figure 19. Send window read in `sysctl`.

The returned values of both keys (`net.ipv4.tcp_rmem` and `net.ipv4.tcp_wmem`) are measured in bytes. The first number represents the minimum buffer size that is used by each TCP socket. The middle one is the default buffer which is allocated when applications create a TCP socket. The last one is the maximum receive buffer that can be allocated for a TCP socket.

The default values used by Linux are:

- Minimum: 10,240
- Default: 87,380
- Maximum: 16,777,216

In the previous test (10 Gbps, 20ms delay), the buffer size was not modified on end-hosts. The BDP for the above test is:

$$\text{BDP} = (10 \cdot 10^9) \cdot (20 \cdot 10^{-3}) = 200,000,000 \text{ bits} = 25,000,000 \text{ bytes} \approx 25 \text{ Mbytes.}$$

Note that this value is significantly greater than the maximum buffer size (16 Mbytes), and therefore, the pipe is not getting filled, which leads to network resources underutilization. Moreover, since Linux systems by default uses half of the send/receive TCP buffers for internal kernel structures (see Section 1.3 Linux systems configuration), only half of the buffer size is used to store TCP segments. Figure 20 shows the calculated window size of a sample packet of the previous test- approximately 8 Mbytes. This is 50% of the default buffer size used by Linux (16 Mbytes).

```

Window size value: 16129
[Calculated window size: 8258048]
[Window size scaling factor: 512]

```

Figure 20. Sample window size from previous test.

Note that the observation in Figure 20 reinforces the best practice described in Section 1.3: in Linux systems, the TCP buffer size must be at least twice the BDP.

4 Modifying buffer size and throughput test

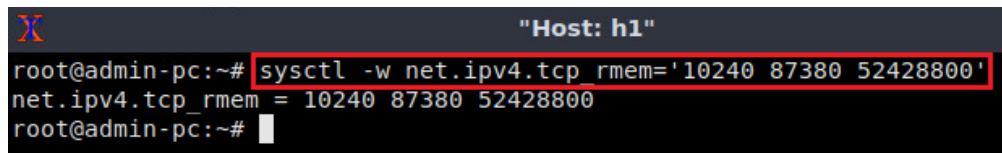
This section repeats the throughput test of Section 4 after modifying the buffer size according to the formula describe above. This test assumes the same network parameters introduced in the previous test, therefore, the bandwidth is limited to 10 Gbps, and the RTT (delay or latency) is 20ms. The send and receive buffer sizes should be set to at least $2 \cdot \text{BDP}$ (if BBR is used as the congestion control algorithm, this should be set to even a larger value, as described in Section 1). We will use 25 Mbytes value for the BDP instead of 25,000,000 bytes ($1 \text{ Mbyte} = 1024^2 \text{ bytes}$).

$$\text{BDP} = 25 \text{ Mbytes} = 25 \cdot 1024^2 \text{ bytes} = 26,214,400 \text{ bytes}$$

$$\text{TCP buffer size} = 2 \cdot \text{BDP} = 2 \cdot 26,214,400 \text{ bytes} = 52,428,800 \text{ bytes}$$

Step 1. To change the TCP receive receive-window size value(s), use the following command on h1's terminal. The values set are: 10,240 (minimum), 87,380 (default), and 52,428,800 (maximum, calculated by doubling the BDP).

```
sysctl -w net.ipv4.tcp_rmem='10240 87380 52428800'
```



```


X                                     "Host: h1"
root@admin-pc:~# sysctl -w net.ipv4.tcp_rmem='10240 87380 52428800'
net.ipv4.tcp_rmem = 10240 87380 52428800
root@admin-pc:~#
  
```

Figure 21. Receive window change in `sysctl`.

The returned values are measured in bytes. 10,240 represents the minimum buffer size that is used by each TCP socket. 87,380 is the default buffer which is allocated when applications create a TCP socket. 52,428,800 is the maximum receive buffer that can be allocated for a TCP socket.

Step 2. To change the current send-window size value(s), use the following command on h1's terminal. The values set are: 10,240 (minimum), 87,380 (default), and 52,428,800 (maximum, calculated by doubling the BDP).

```
sysctl -w net.ipv4.tcp_wmem='10240 87380 52428800'
```



```

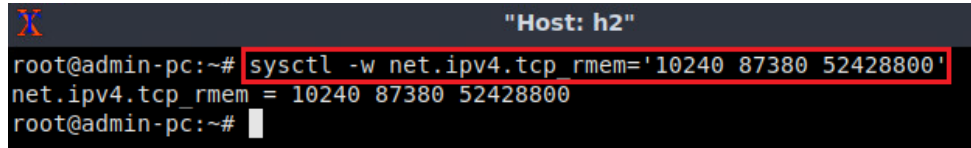
X                                     "Host: h1"
root@admin-pc:~# sysctl -w net.ipv4.tcp_wmem='10240 87380 52428800'
net.ipv4.tcp_wmem = 10240 87380 52428800
root@admin-pc:~#
  
```

Figure 22. Send window change in `sysctl`.

Next, the same commands must be configured on host h2.

Step 3. To change the current receiver-window size value(s), use the following command on h2's terminal. The values set are: 10,240 (minimum), 87,380 (default), and 52,428,800 (maximum, calculated by doubling the BDP).

```
sysctl -w net.ipv4.tcp_rmem='10240 87380 52428800'
```



```

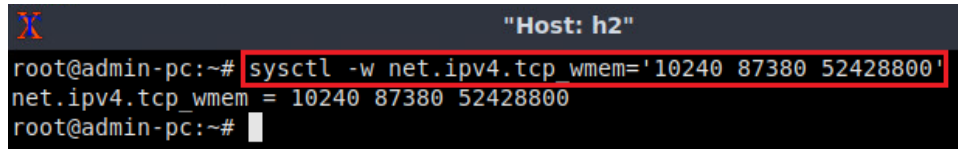
X "Host: h2"
root@admin-pc:~# sysctl -w net.ipv4.tcp_rmem='10240 87380 52428800'
net.ipv4.tcp_rmem = 10240 87380 52428800
root@admin-pc:~#

```

Figure 23. Receive window change in `sysctl`.

Step 4. To change the current send-window size value(s), use the following command on h2's terminal. The values set are: 10,240 (minimum), 87,380 (default), and 52,428,800 (maximum, calculated by doubling the BDP).

```
sysctl -w net.ipv4.tcp_wmem='10240 87380 52428800'
```



```

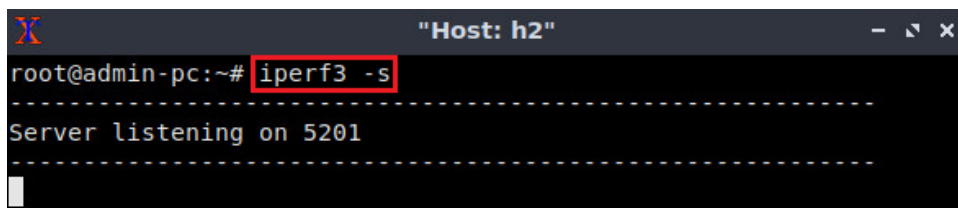
X "Host: h2"
root@admin-pc:~# sysctl -w net.ipv4.tcp_wmem='10240 87380 52428800'
net.ipv4.tcp_wmem = 10240 87380 52428800
root@admin-pc:~#

```

Figure 24. Send window change in `sysctl`.

Step 5. The user can now verify the rate limit configuration by using the `iperf3` tool to measure throughput. To launch iPerf3 in server mode, run the command `iperf3 -s` in h2's terminal:

```
iperf3 -s
```



```

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

```

Figure 25. Host h2 running iPerf3 as server.

Step 6. Now to launch iPerf3 in client mode again by running the command `iperf3 -c [10.0.0.2]` in h1's terminal:

```
iperf3 -c 10.0.0.2
```



```

Host: h1
root@admin-pc:~# iperf3 -c 10.0.0.2
Connecting to host 10.0.0.2, port 5201
[ 15] local 10.0.0.1 port 47094 connected to 10.0.0.2 port 5201
[ ID] Interval           Transfer     Bitrate      Retr  Cwnd
[ 15]  0.00-1.00    sec     925 MBytes  7.76 Gbits/sec    45   39.8 MBytes
[ 15]  1.00-2.00    sec    1.11 GBytes  9.57 Gbits/sec     0   39.8 MBytes
[ 15]  2.00-3.00    sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
[ 15]  3.00-4.00    sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
[ 15]  4.00-5.00    sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
[ 15]  5.00-6.00    sec    1.11 GBytes  9.55 Gbits/sec     0   39.8 MBytes
[ 15]  6.00-7.00    sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
[ 15]  7.00-8.00    sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
[ 15]  8.00-9.00    sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
[ 15]  9.00-10.00   sec    1.11 GBytes  9.56 Gbits/sec     0   39.8 MBytes
-----
[ ID] Interval           Transfer     Bitrate      Retr
[ 15]  0.00-10.00    sec    10.9 GBytes  9.38 Gbits/sec    45
[ 15]  0.00-10.04    sec    10.9 GBytes  9.34 Gbits/sec
                                     sender
                                     receiver

iperf Done.
root@admin-pc:~#

```

Figure 26. iPerf3 throughput test.

Note the measured throughput now is approximately 10 Gbps, which is close to the value assigned in our `tbfb` rule (10 Gbps).

This concludes Lab 8. Stop the emulation and then exit out of MiniEdit and Linux terminal.

References

1. 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.
2. K. Fall, S. Floyd, "Simulation-based comparisons of tahoe, reno, and sack TCP," *Computer Communication Review*, vol. 26, issue 3, Jul. 1996.
3. 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.
4. 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>
5. V. Jacobson, R. Braden, D. Borman, "TCP extensions for high performance," *Internet Request for Comments, RFC Edit, RFC 1323*, May 1992. [Online]. Available: <https://tools.ietf.org/rfc/rfc1323.txt>