

CN Assignment - 2

Name: Rachit Verma

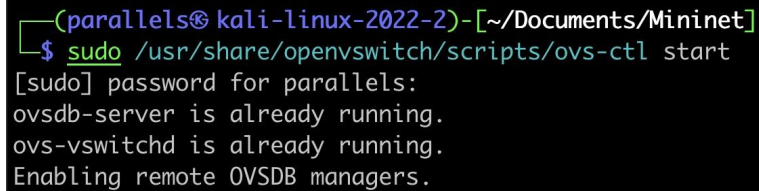
Roll no: 21110171

Name: Srujan Kumar Shetty

Roll. no: 21110214

Command to setup ovs:

```
sudo /usr/share/openvswitch/scripts/ovs-ctl start
```



```
(parallels@kali-linux-2022-2)-[~/Documents/Mininet]
$ sudo /usr/share/openvswitch/scripts/ovs-ctl start
[sudo] password for parallels:
ovsdb-server is already running.
ovs-vswitchd is already running.
Enabling remote OVSDb managers.
```

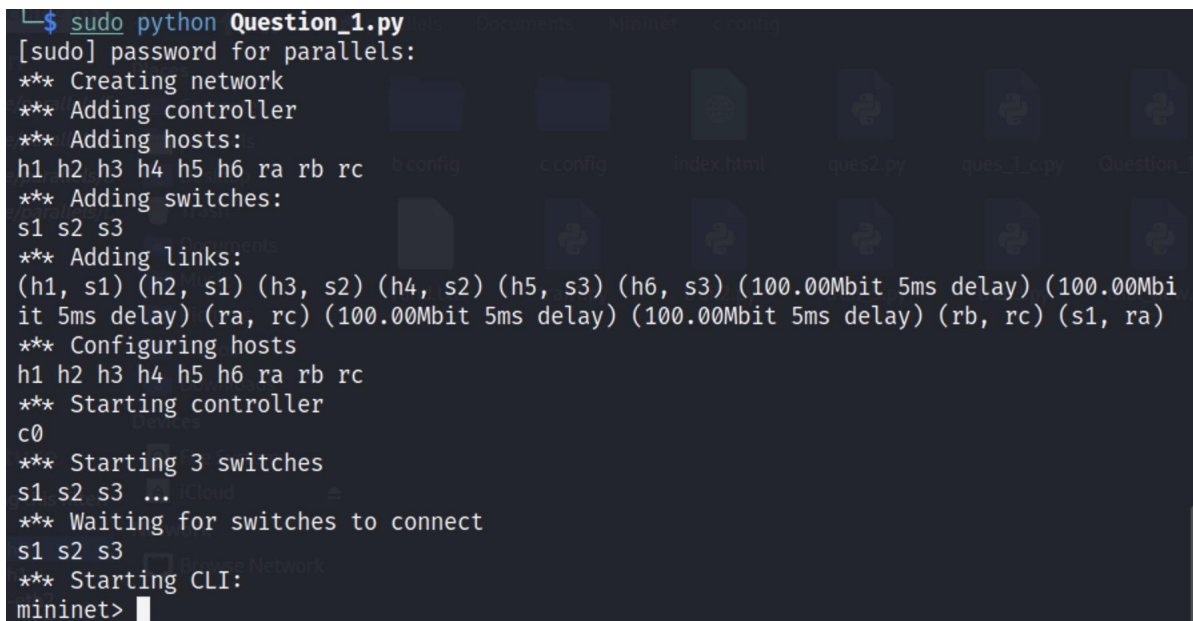
Fig. OVS setup

Question 1

Command to run the topology for question1:

```
sudo python Question_1.py
```

Note: This also opens the mininet CLI for this topology.



```
$ sudo python Question_1.py
[sudo] password for parallels:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 ra rb rc
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (100.00Mbit 5ms delay) (100.00Mbit 5ms delay) (ra, rc) (100.00Mbit 5ms delay) (100.00Mbit 5ms delay) (rb, rc) (s1, ra)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 ra rb rc
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Waiting for switches to connect
s1 s2 s3
*** Starting CLI:
mininet>
```

Fig. Example working of the program

Note: The links connecting ra<->rb, rb<->rc and ra<->rc have parameters: Bandwidth = 100Mbps and delay = 5ms. This has been done to get realistic delays [useful when checking the latency difference between different routes].

Proof of working:

1. Pingall test:

Tests ping reachability across all the connections.

```
mininet> pingall
*** Ping: testing ping reachability
h1 -> h2 h3 h4 h5 h6 ra rb rc
h2 -> h1 h3 h4 h5 h6 ra rb rc
h3 -> h1 h2 h4 h5 h6 ra rb rc
h4 -> h1 h2 h3 h5 h6 ra rb rc
h5 -> h1 h2 h3 h4 h6 ra rb rc
h6 -> h1 h2 h3 h4 h5 ra rb rc
ra -> h1 h2 h3 h4 h5 h6 rb rc
rb -> h1 h2 h3 h4 h5 h6 ra rc
rc -> h1 h2 h3 h4 h5 h6 ra rb
*** Results: 0% dropped (72/72 received)
```

Fig. Successful pingall test

2. Traceroute test:

```
mininet> h1 traceroute h4
traceroute to 10.100.0.200 (10.100.0.200), 30 hops max, 60 byte packets
 1  10.0.0.1 (10.0.0.1)  1.939 ms  1.693 ms  2.820 ms
 2  10.0.50.2 (10.0.50.2) 23.800 ms 23.655 ms 23.674 ms
 3  10.100.0.200 (10.100.0.200) 26.442 ms 26.368 ms 26.320 ms
```

-> Router ra
-> If: rb-eth1 [rb]
-> Host h4

```
mininet> h1 traceroute h6
traceroute to 10.200.0.200 (10.200.0.200), 30 hops max, 60 byte packets
 1  10.0.0.1 (10.0.0.1)  2.281 ms  2.193 ms  2.176 ms
 2  10.0.150.2 (10.0.150.2) 13.443 ms 13.428 ms 13.391 ms
 3  10.200.0.200 (10.200.0.200) 13.542 ms 13.579 ms 13.676 ms
```

-> Router ra
-> If: rc-eth2 [rc]
-> Host h6

```
mininet> h1 traceroute h2
traceroute to 10.0.0.200 (10.0.0.200), 30 hops max, 60 byte packets
 1  10.0.0.200 (10.0.0.200) 5.105 ms 4.233 ms 4.219 ms
```

-> Host h2

```
mininet> h3 traceroute h5
traceroute to 10.200.0.100 (10.200.0.100), 30 hops max, 60 byte packets
 1  10.100.0.1 (10.100.0.1) 6.068 ms 5.942 ms 5.769 ms
 2  10.0.100.2 (10.0.100.2) 27.209 ms 27.180 ms 27.183 ms
 3  10.200.0.100 (10.200.0.100) 32.688 ms 32.661 ms 32.650 ms
```

-> Router rb
-> If: rc-eth1 [rc]
-> Host h5

Note: IF: stands for Interface.

Part b

Upon using the command in the mininet CLI:

```
ra wireshark
```

The wireshark window opened now is based on the router ra and can specifically sniff ra's interfaces [a local configuration]. We can choose 'any' interface from the interface list to sniff all the packets through router 'ra', that is through all its interfaces.

The saved pcap can be found in the repository with the filename 'Question_1_b.pcap'.

Pcap contains the following packets:

1. pingall
2. h1 ping h5
3. h1 ping h3

The other way to accomplish this would be using tcpdump and store the received packets to a pcap file. Again, there are several ways to automate this process.

Part c

Command Sequence to alter the Routing:

Steps:

1. Open xterm on ra. Using:

```
xterm ra
```
2. Route the packets belonging to 10.200.0.0/24 subnet to rb using:

```
ip route change 10.200.0.0/24 via 10.0.50.2 dev ra-eth1
```

Additionally we can use command 'ip route show' to confirm the changes.
[Routing table details provided in Part D]

3. Similarly, open xterm on rc. Using:

```
xterm rc
```
4. Route the packets belonging to 10.0.0.0/24 subnet to rb using:

```
ip route change 10.0.0.0/24 via 10.0.100.1 dev rc-eth1
```

Additionally we can use command 'ip route show' to confirm the changes.
[Routing table details provided in Part D]

Ping[RTT] measurements:

Default routing:

Test 1: h1 ping h6

```
mininet> h1 ping h6 -c 5
PING 10.200.0.200 (10.200.0.200) 56(84) bytes of data.
64 bytes from 10.200.0.200: icmp_seq=1 ttl=62 time=14.8 ms
64 bytes from 10.200.0.200: icmp_seq=2 ttl=62 time=14.0 ms
64 bytes from 10.200.0.200: icmp_seq=3 ttl=62 time=12.8 ms
64 bytes from 10.200.0.200: icmp_seq=4 ttl=62 time=11.7 ms
64 bytes from 10.200.0.200: icmp_seq=5 ttl=62 time=11.7 ms

— 10.200.0.200 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4016ms
rtt min/avg/max/mdev = 11.698/13.008/14.807/1.234 ms
```

Fig. Ping results of test 1

Test 2: h2 ping h4

```
mininet> h2 ping h4 -c 5
PING 10.100.0.200 (10.100.0.200) 56(84) bytes of data.
64 bytes from 10.100.0.200: icmp_seq=1 ttl=62 time=26.5 ms
64 bytes from 10.100.0.200: icmp_seq=2 ttl=62 time=12.8 ms
64 bytes from 10.100.0.200: icmp_seq=3 ttl=62 time=11.6 ms
64 bytes from 10.100.0.200: icmp_seq=4 ttl=62 time=12.1 ms
64 bytes from 10.100.0.200: icmp_seq=5 ttl=62 time=12.9 ms

— 10.100.0.200 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4026ms
rtt min/avg/max/mdev = 11.646/15.199/26.527/5.683 ms
```

Fig. Ping results of test 2

Test 3: h5 ping h3

```
mininet> h5 ping h3 -c 5
PING 10.100.0.100 (10.100.0.100) 56(84) bytes of data.
64 bytes from 10.100.0.100: icmp_seq=1 ttl=62 time=26.5 ms
64 bytes from 10.100.0.100: icmp_seq=2 ttl=62 time=13.9 ms
64 bytes from 10.100.0.100: icmp_seq=3 ttl=62 time=12.9 ms
64 bytes from 10.100.0.100: icmp_seq=4 ttl=62 time=12.1 ms
64 bytes from 10.100.0.100: icmp_seq=5 ttl=62 time=12.1 ms

— 10.100.0.100 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4016ms
rtt min/avg/max/mdev = 12.072/15.478/26.468/5.534 ms
```

Fig. Ping results for test 3.

Test 4: h5 ping h3

```
mininet> h5 ping h3 -c 4
PING 10.100.0.100 (10.100.0.100) 56(84) bytes of data.
64 bytes from 10.100.0.100: icmp_seq=1 ttl=62 time=11.0 ms
64 bytes from 10.100.0.100: icmp_seq=2 ttl=62 time=12.9 ms
64 bytes from 10.100.0.100: icmp_seq=3 ttl=62 time=11.9 ms
64 bytes from 10.100.0.100: icmp_seq=4 ttl=62 time=12.3 ms

— 10.100.0.100 ping statistics —
4 packets transmitted, 4 received, 0% packet loss, time 3010ms
rtt min/avg/max/mdev = 10.978/12.002/12.870/0.685 ms
```

Fig. Ping results for test 3.

Modified routing:

Test 1: h1 ping h6

```
mininet> h1 ping h6 -c 5
PING 10.200.0.200 (10.200.0.200) 56(84) bytes of data.
64 bytes from 10.200.0.200: icmp_seq=1 ttl=61 time=24.6 ms
64 bytes from 10.200.0.200: icmp_seq=2 ttl=61 time=22.3 ms
64 bytes from 10.200.0.200: icmp_seq=3 ttl=61 time=21.1 ms
64 bytes from 10.200.0.200: icmp_seq=4 ttl=61 time=21.0 ms
64 bytes from 10.200.0.200: icmp_seq=5 ttl=61 time=21.7 ms

— 10.200.0.200 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4006ms
rtt min/avg/max/mdev = 21.004/22.132/24.576/1.310 ms
```

Fig. Results of test 1

Here we can see that the average rtt has almost doubled compared to the default routing.

Test 2: h2 ping h4

```
mininet> h2 ping h4 -c 5
PING 10.100.0.200 (10.100.0.200) 56(84) bytes of data.
64 bytes from 10.100.0.200: icmp_seq=1 ttl=62 time=11.0 ms
64 bytes from 10.100.0.200: icmp_seq=2 ttl=62 time=12.9 ms
64 bytes from 10.100.0.200: icmp_seq=3 ttl=62 time=12.7 ms
64 bytes from 10.100.0.200: icmp_seq=4 ttl=62 time=11.7 ms
64 bytes from 10.100.0.200: icmp_seq=5 ttl=62 time=12.9 ms

— 10.100.0.200 ping statistics —
5 packets transmitted, 5 received, 0% packet loss, time 4018ms
rtt min/avg/max/mdev = 10.996/12.251/12.932/0.770 ms
```

Fig. Results of test 2

Here we can see that the average rtt has remained same compared to the default routing.

Test 3: h5 ping h3

```
mininet> h5 ping h3 -c 4
PING 10.100.0.100 (10.100.0.100) 56(84) bytes of data.
64 bytes from 10.100.0.100: icmp_seq=1 ttl=62 time=11.5 ms
64 bytes from 10.100.0.100: icmp_seq=2 ttl=62 time=12.9 ms
64 bytes from 10.100.0.100: icmp_seq=3 ttl=62 time=11.6 ms
64 bytes from 10.100.0.100: icmp_seq=4 ttl=62 time=11.8 ms

— 10.100.0.100 ping statistics —
4 packets transmitted, 4 received, 0% packet loss, time 3006ms
rtt min/avg/max/mdev = 11.467/11.930/12.853/0.543 ms
```

Fig. Results of test 3

Here we can see that the average rtt has remained same compared to the default routing.

Test 4: h5 ping h3

```
mininet> h5 ping h2 -c 4
PING 10.0.0.200 (10.0.0.200) 56(84) bytes of data.
64 bytes from 10.0.0.200: icmp_seq=1 ttl=61 time=25.7 ms
64 bytes from 10.0.0.200: icmp_seq=2 ttl=61 time=25.6 ms
64 bytes from 10.0.0.200: icmp_seq=3 ttl=61 time=25.6 ms
64 bytes from 10.0.0.200: icmp_seq=4 ttl=61 time=24.1 ms

— 10.0.0.200 ping statistics —
4 packets transmitted, 4 received, 0% packet loss, time 3018ms
rtt min/avg/max/mdev = 24.149/25.270/25.718/0.649 ms
```

Fig. Results of test 4

Here we can see that the average rtt has almost doubled compared to the default routing.

IPerf Measurements:

Default routing:

Configuration: A TCP server is set up at h6 [IP: 10.200.0.200] at port 5002

Test 1: Traffic generated with a TCP client setup at h1 directed to h6.

```
mininet> h1 iperf -c 10.200.0.200 -t 5 -p 5002

Client connecting to 10.200.0.200, TCP port 5002
TCP window size: 85.3 KByte (default)

[ 1] local 10.0.0.100 port 43508 connected with 10.200.0.200 port 5002 (icwnd/mss/irrtt=14/1448/13917)
[ ID] Interval      Transfer      Bandwidth
[ 1] 0.0000-5.9743 sec 66.5 MBytes  93.4 Mbits/sec
```

Fig. Results of test 1

Test 2: Traffic generated with a TCP client setup at h4 directed to h6.

```
mininet> h4 iperf -c h6 -t 5 -p 5002
Client connecting to 10.200.0.200, TCP port 5002
TCP window size: 85.3 KByte (default)
[ 1] local 10.100.0.200 port 34504 connected with 10.200.0.200 port 5002 (icwnd/mss/irrt=14/1448/27406)
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-6.3865 sec 70.5 MBytes 92.6 Mbits/sec
```

Fig. Results of test 2

Modified:

Configuration: A TCP server is set up at h6 [IP: 10.200.0.200] at port 4000

Test 1: Traffic generated with a TCP client setup at h1 directed to h6.

```
mininet> h1 iperf -c 10.200.0.200 -t 5 -p 4000
Client connecting to 10.200.0.200, TCP port 4000
TCP window size: 85.3 KByte (default)
[ 1] local 10.0.0.100 port 52632 connected with 10.200.0.200 port 4000 (icwnd/mss/irrt=14/1448/25277)
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-6.3639 sec 69.8 MBytes 91.9 Mbits/sec
```

Fig. Results of test 1

Here we can see that the throughput has slightly decreased compared to the default routing.

Test 2: Traffic generated with a TCP client setup at h4 directed to h6.

```
mininet> h4 iperf -c 10.200.0.200 -t 5 -p 4000
Client connecting to 10.200.0.200, TCP port 4000
TCP window size: 85.3 KByte (default)
[ 1] local 10.100.0.200 port 55432 connected with 10.200.0.200 port 4000 (icwnd/mss/irrt=14/1448/15567)
[ ID] Interval      Transfer    Bandwidth
[ 1] 0.0000-6.2961 sec 70.0 MBytes 93.3 Mbits/sec
```

Fig. Results of test 2

Here we can see that the throughput has remained same compared to the default routing.

Part d

With default routing:

IP Routing Table for Router ra:

```
mininet> ra route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0         0.0.0.0        255.255.255.0   U        0      0      0 ra-eth0
10.0.50.0        0.0.0.0        255.255.255.0   U        0      0      0 ra-eth1
10.0.150.0       0.0.0.0        255.255.255.0   U        0      0      0 ra-eth2
10.100.0.0       10.0.50.2      255.255.255.0   UG       0      0      0 ra-eth1
10.200.0.0       10.0.150.2     255.255.255.0   UG       0      0      0 ra-eth2
```

```

# ip route show
10.0.0.0/24 dev ra-eth0 proto kernel scope link src 10.0.0.1
10.0.50.0/24 dev ra-eth1 proto kernel scope link src 10.0.50.1
10.0.150.0/24 dev ra-eth2 proto kernel scope link src 10.0.150.1
10.100.0.0/24 via 10.0.50.2 dev ra-eth1
10.200.0.0/24 via 10.0.150.2 dev ra-eth2

```

IP Routing Table for Router rb:

```
mininet> rb route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0         10.0.50.1      255.255.255.0   UG       0      0      0 rb-eth1
10.0.50.0        0.0.0.0        255.255.255.0   U        0      0      0 rb-eth1
10.0.100.0       0.0.0.0        255.255.255.0   U        0      0      0 rb-eth2
10.100.0.0       0.0.0.0        255.255.255.0   U        0      0      0 rb-eth0
10.200.0.0       10.0.100.2     255.255.255.0   UG       0      0      0 rb-eth2
```

```

# ip route
10.0.0.0/24 via 10.0.50.1 dev rb-eth1
10.0.50.0/24 dev rb-eth1 proto kernel scope link src 10.0.50.2
10.0.100.0/24 dev rb-eth2 proto kernel scope link src 10.0.100.1
10.100.0.0/24 dev rb-eth0 proto kernel scope link src 10.100.0.1
10.200.0.0/24 via 10.0.100.2 dev rb-eth2

```

IP Routing Table for Router rc:

```
mininet> rc route
Kernel IP routing table
Destination      Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0         10.0.150.1     255.255.255.0   UG       0      0      0 rc-eth2
10.0.100.0       0.0.0.0        255.255.255.0   U        0      0      0 rc-eth1
10.0.150.0       0.0.0.0        255.255.255.0   U        0      0      0 rc-eth2
10.100.0.0       10.0.100.1     255.255.255.0   UG       0      0      0 rc-eth1
10.200.0.0       0.0.0.0        255.255.255.0   U        0      0      0 rc-eth0
```

```

# ip route
10.0.0.0/24 via 10.0.150.1 dev rc-eth2
10.0.100.0/24 dev rc-eth1 proto kernel scope link src 10.0.100.2
10.0.150.0/24 dev rc-eth2 proto kernel scope link src 10.0.150.2
10.100.0.0/24 via 10.0.100.1 dev rc-eth1
10.200.0.0/24 dev rc-eth0 proto kernel scope link src 10.200.0.1

```

With modified routing: [ra->rb->rc instead of ra->rc and rc->rb->ra instead of rc->ra]

IP Routing Table for Router ra:

Kernel IP routing table							
Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	0.0.0.0	255.255.255.0	U	0	0	0	ra-eth0
10.0.50.0	0.0.0.0	255.255.255.0	U	0	0	0	ra-eth1
10.0.150.0	0.0.0.0	255.255.255.0	U	0	0	0	ra-eth2
10.100.0.0	10.0.50.2	255.255.255.0	UG	0	0	0	ra-eth1
10.200.0.0	10.0.50.2	255.255.255.0	UG	0	0	0	ra-eth1

```
(root@kali-linux-2022-2) - [~/home/parallects/Documents/Mininet]
# ip route show
10.0.0.0/24 dev ra-eth0 proto kernel scope link src 10.0.0.1
10.0.50.0/24 dev ra-eth1 proto kernel scope link src 10.0.50.1
10.0.150.0/24 dev ra-eth2 proto kernel scope link src 10.0.150.1
10.100.0.0/24 via 10.0.50.2 dev ra-eth1
10.200.0.0/24 via 10.0.50.2 dev ra-eth1
```

IP Routing Table for Router rb:

Kernel IP routing table							
Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	10.0.50.1	255.255.255.0	UG	0	0	0	rb-eth1
10.0.50.0	0.0.0.0	255.255.255.0	U	0	0	0	rb-eth1
10.0.100.0	0.0.0.0	255.255.255.0	U	0	0	0	rb-eth2
10.100.0.0	0.0.0.0	255.255.255.0	U	0	0	0	rb-eth0
10.200.0.0	10.0.100.2	255.255.255.0	UG	0	0	0	rb-eth2

```
10.0.0.0/24 via 10.0.50.1 dev rb-eth1
10.0.50.0/24 dev rb-eth1 proto kernel scope link src 10.0.50.2
10.0.100.0/24 dev rb-eth2 proto kernel scope link src 10.0.100.1
10.100.0.0/24 dev rb-eth0 proto kernel scope link src 10.100.0.1
10.200.0.0/24 via 10.0.100.2 dev rb-eth2
```

IP Routing Table for Router rc:

Kernel IP routing table							
Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	10.0.100.1	255.255.255.0	UG	0	0	0	rc-eth1
10.0.100.0	0.0.0.0	255.255.255.0	U	0	0	0	rc-eth1
10.0.150.0	0.0.0.0	255.255.255.0	U	0	0	0	rc-eth2
10.100.0.0	10.0.100.1	255.255.255.0	UG	0	0	0	rc-eth1
10.200.0.0	0.0.0.0	255.255.255.0	U	0	0	0	rc-eth0

```
10.0.0.0/24 via 10.0.100.1 dev rc-eth1
10.0.100.0/24 dev rc-eth1 proto kernel scope link src 10.0.100.2
10.0.150.0/24 dev rc-eth2 proto kernel scope link src 10.0.150.2
10.100.0.0/24 via 10.0.100.1 dev rc-eth1
10.200.0.0/24 dev rc-eth0 proto kernel scope link src 10.200.0.1
```

Question 2

Important comment

Server and client work reversibly in iperf as compared to the practical scenarios. Hence the clients are the ones that are generating the traffic while the servers are the ones that are sending the acks. As a result, in the c part, hosts 1,2 and 3 are sending packets to host 4, and using this configuration, the throughput has been tested.

Instructions for running the file

Make sure that all the dependencies are installed and set up. Once that has been done, run the following command:

```
sudo python3 Question_2.py -config= -control_algo= -link_loss=
```

Use 'b' or 'c' for the config flags to run the code for the corresponding part.

Use 'cubic', 'bbr', 'vegas', 'reno', to use the respective congestion control algorithms.

Enter the percentage of the link loss for the link_loss parameter.

For example, if you wanted to run c part with vegas as the congestion control algorithm and no link loss, you would run:

```
sudo python3 Question_2.py -config=c -control_algo=vegas -link_loss=0
```

Running this command will automatically set up the whole configuration including the servers, the clients and the tcpdump commands for capturing packets so they can be analyzed later.

When the command is done running, 4 files(in case of part c) and 1 file(in case of part b) will be generated in the working directory. These files will be the pcap files containing the traffic of the corresponding host. capture1.pcap will contain the traffic for the first host, capture2.pcap will contain the traffic for the second host and so on. These generated files can be analyzed later using tools like wireshark. All the graphs that are shown below have been generated using the wireshark tool.

```

*** Done
rachit0206@Wannabe-Macbook:~/Documents/CN Assignment 2$ sudo python3 Q2.py --con
fig=c --control_algo=cubic --link_loss=0
[sudo] password for rachit0206:
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4
*** Adding switches:
s1 s2
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (10.00Mbit 5ms delay 0.00000% loss) (10.00Mb
it 5ms delay 0.00000% loss) (s1, s2)
*** Configuring hosts
h1 h2 h3 h4
*** Starting controller
c0
*** Starting 2 switches
s1 s2 ... (10.00Mbit 5ms delay 0.00000% loss) (10.00Mbit 5ms delay 0.00000% loss)
*** Stopping 1 controllers
c0
*** Stopping 5 links

```

Fig. Example working of the program

Part b and d

Cubic

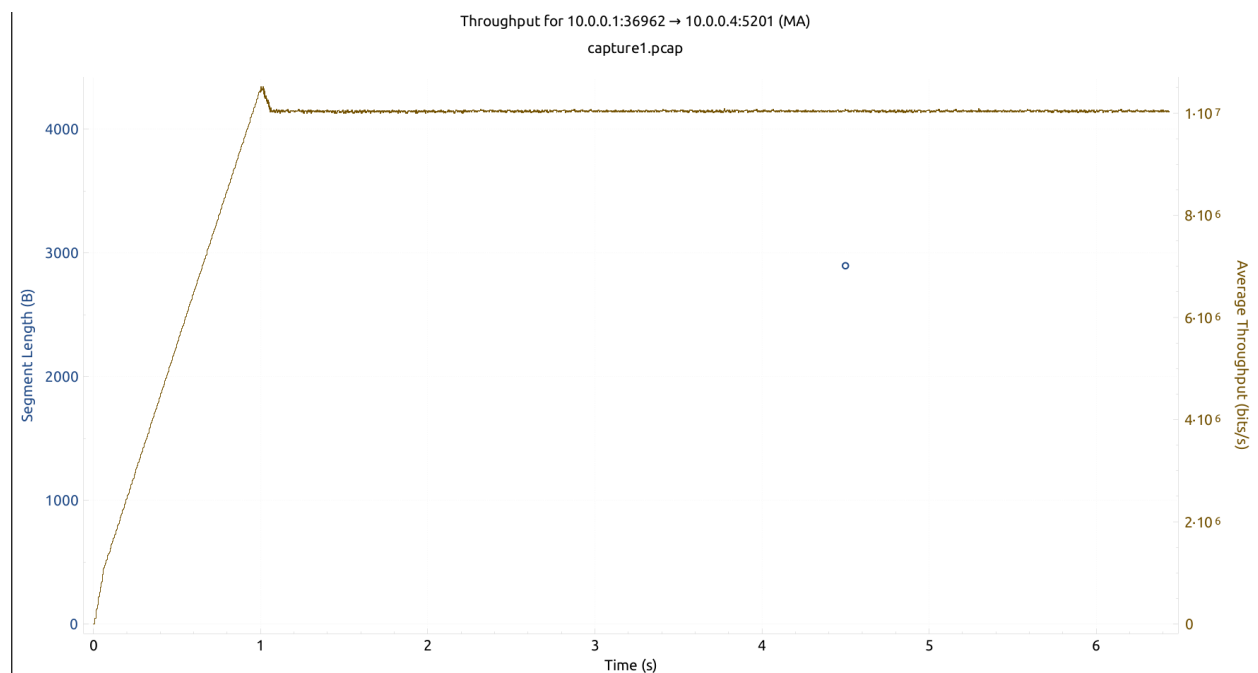


Fig. Throughput graph when link loss was 0%

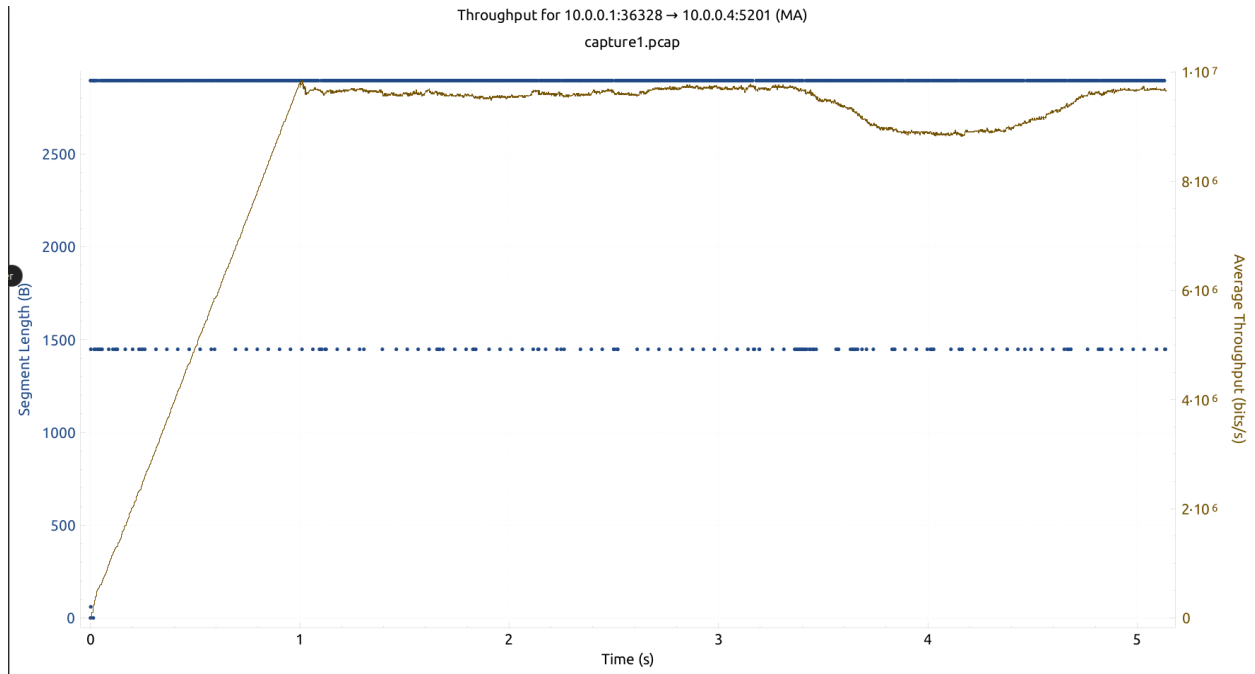


Fig. Throughput graph when link loss was 1%

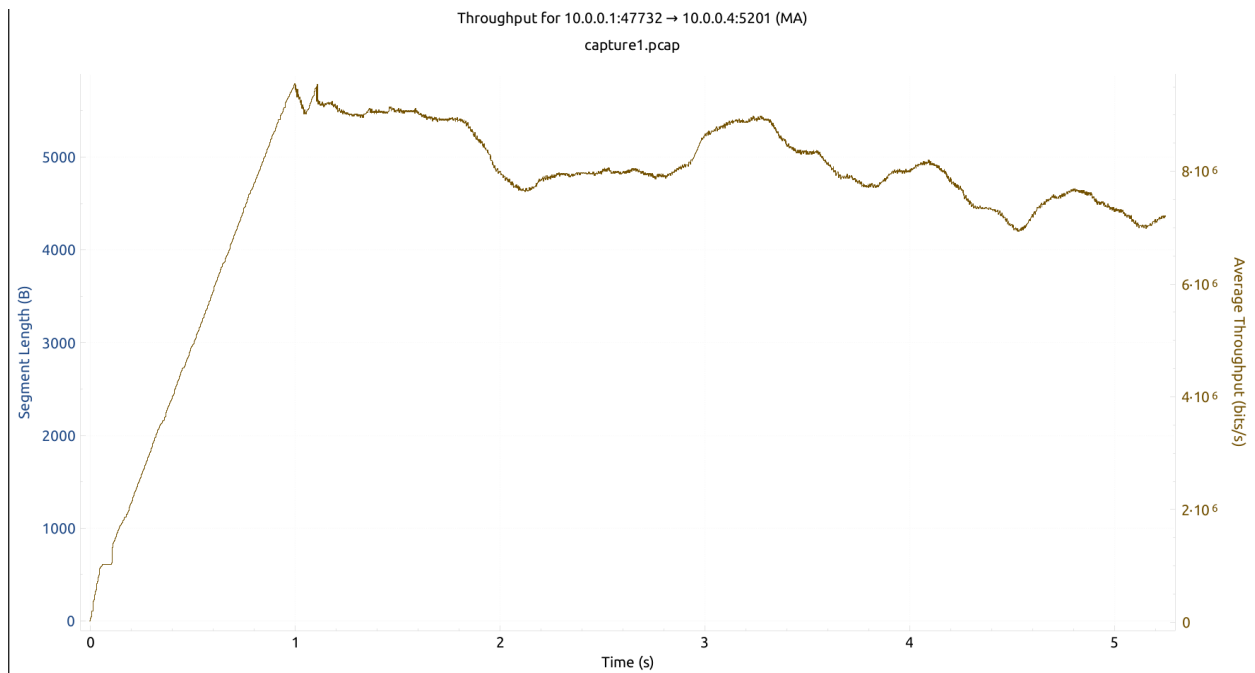


Fig. Throughput when link loss was 2%

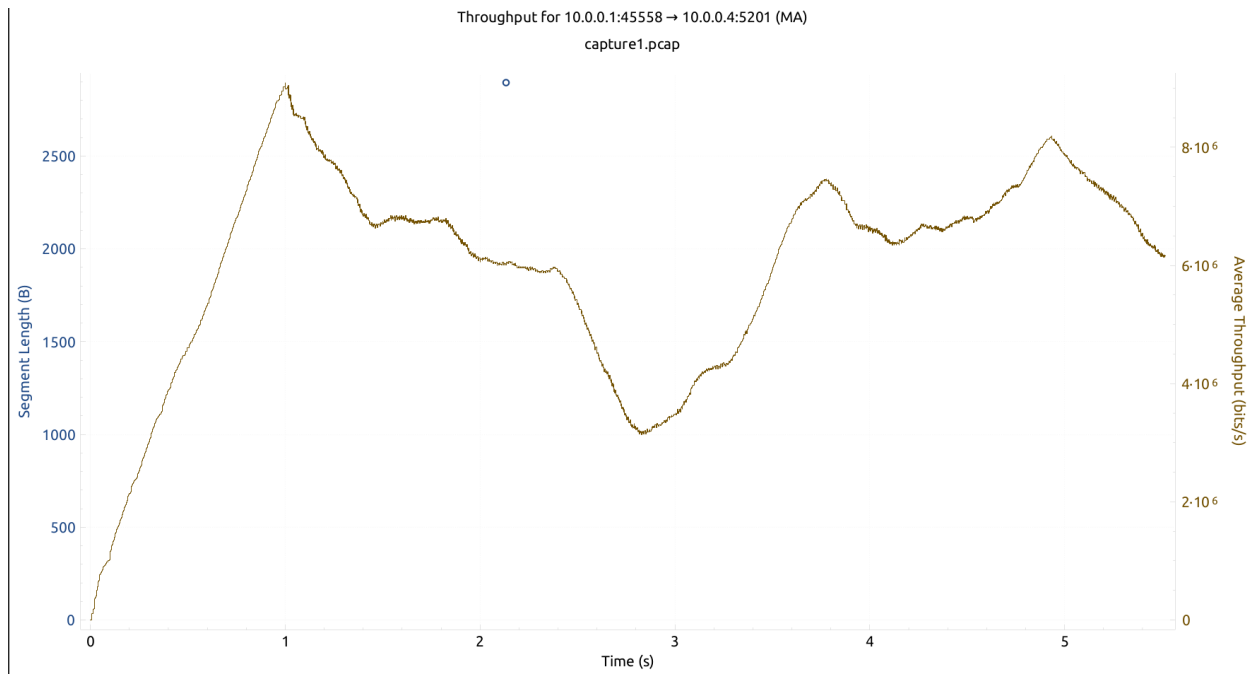


Fig. Throughput when link loss was 3%

As is visible from the figures above, the throughput gets progressively unstable as the link loss is increased. Initially, the network is steadily increasing its throughput, and as it hits a particular threshold, the link loss starts to affect the throughput. The network then needs to recover in accordance with the cubic congestion control algorithm. In various parts of the figures, the characteristics of the cubic congestion control algorithm are seen, with concave upwards and concave downwards parts both visible. As the link loss increases, the lowest throughput obtained also decreases for each figure. With increase in frequency of losses, the congestion control algorithm starts to increase its throughput from lower values.

Reno

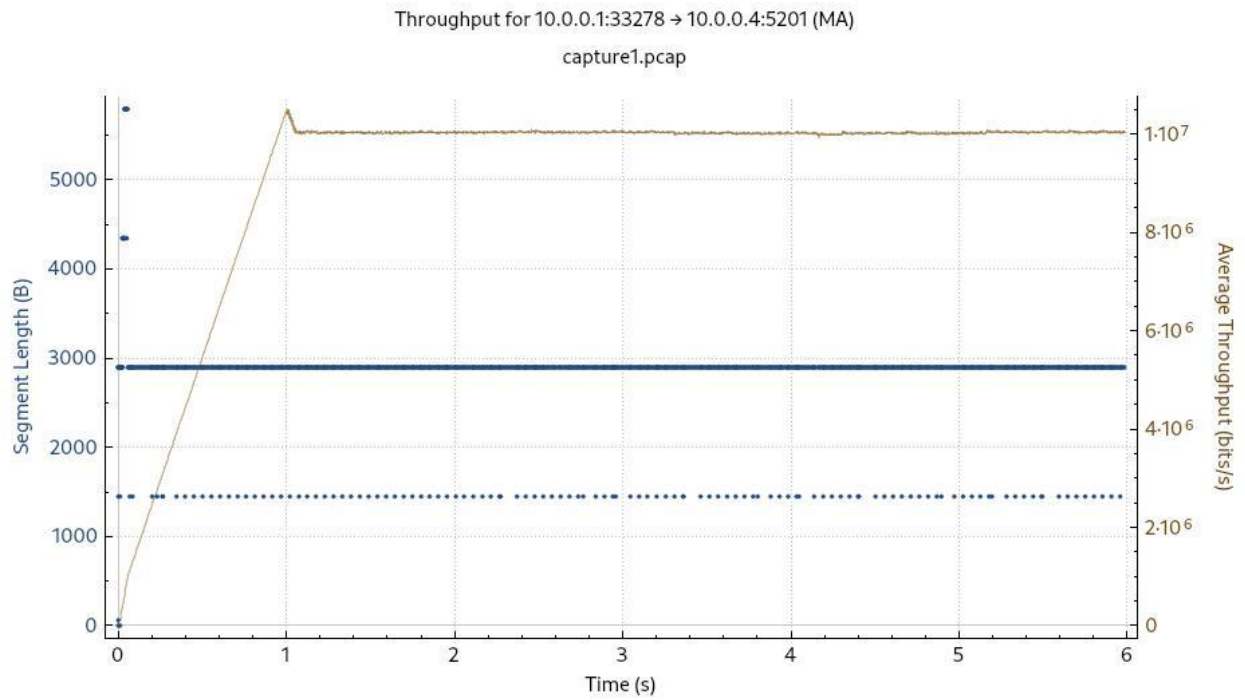


Fig. Throughput when link loss was 0%

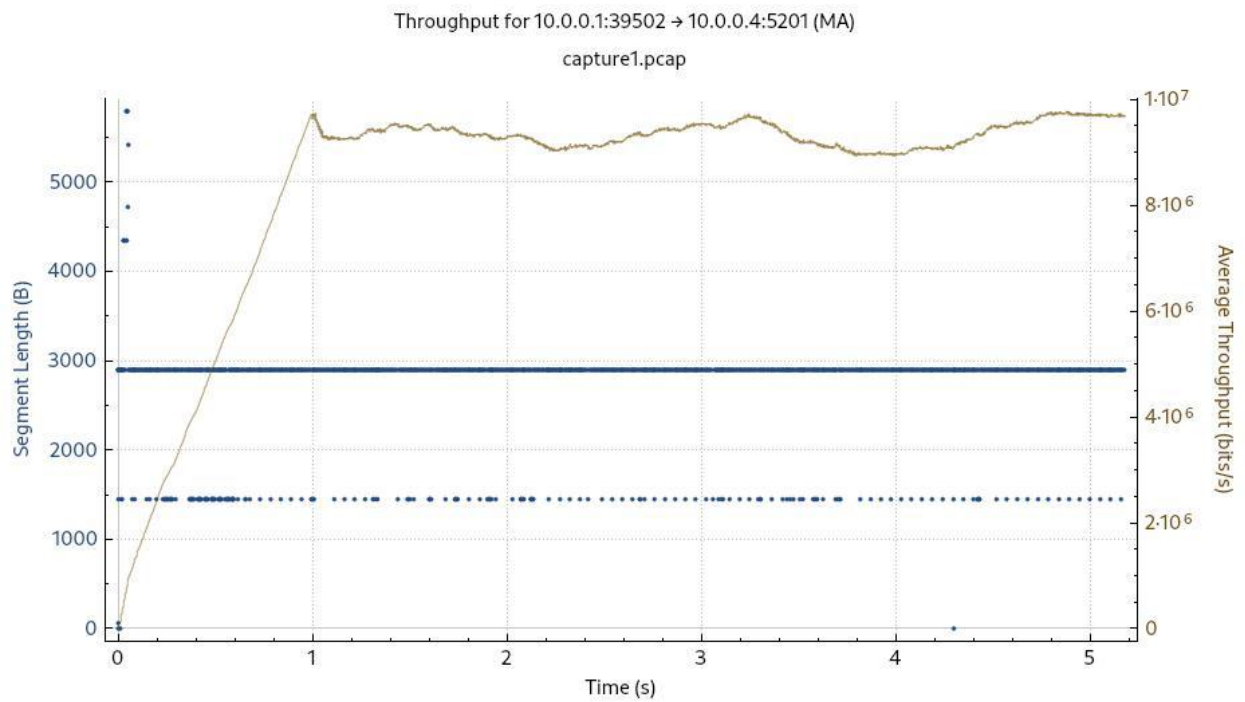


Fig. Throughput when link loss 1%

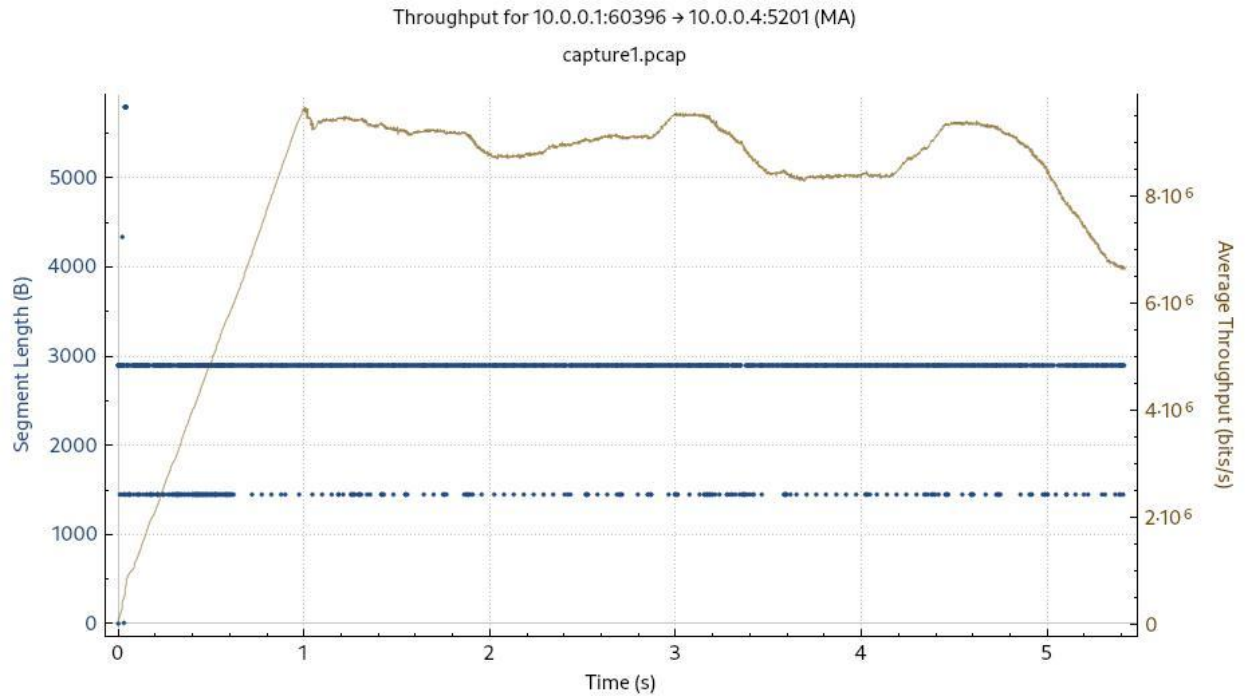


Fig. Throughput when link loss 2%

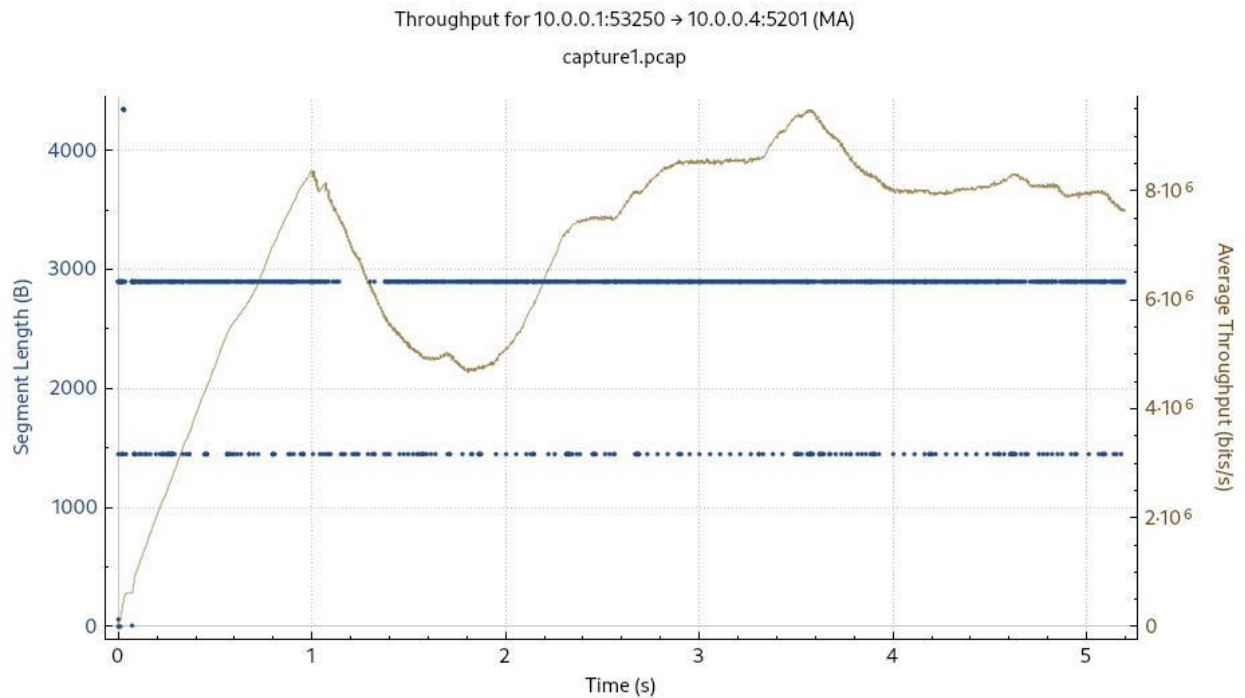


Fig. Throughput when link loss 3%

Again, a similar pattern as before can be observed here, with the throughput becoming less stable as the link loss percentage is increased. The minima obtained in all the graphs is similarly lowest when the link loss parameter was 3%.

In certain parts of the graphs, the characteristics of the reno algorithm can be found, with throughput sometimes falling to much lower values than other usual (in accordance with the type of the congestion event). The slow start and congestion avoidance phases can also be seen in some parts, with the point of change occurring at the appropriate points.

BBR

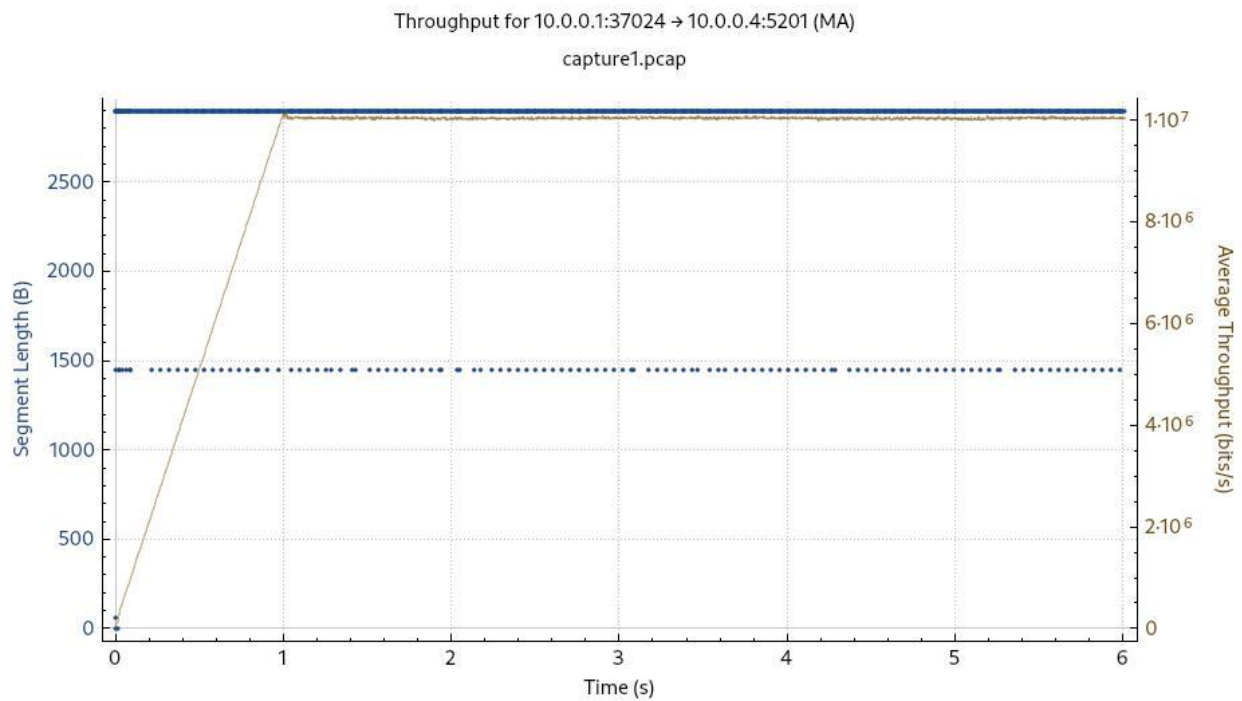


Fig. Throughput when link loss was 0%

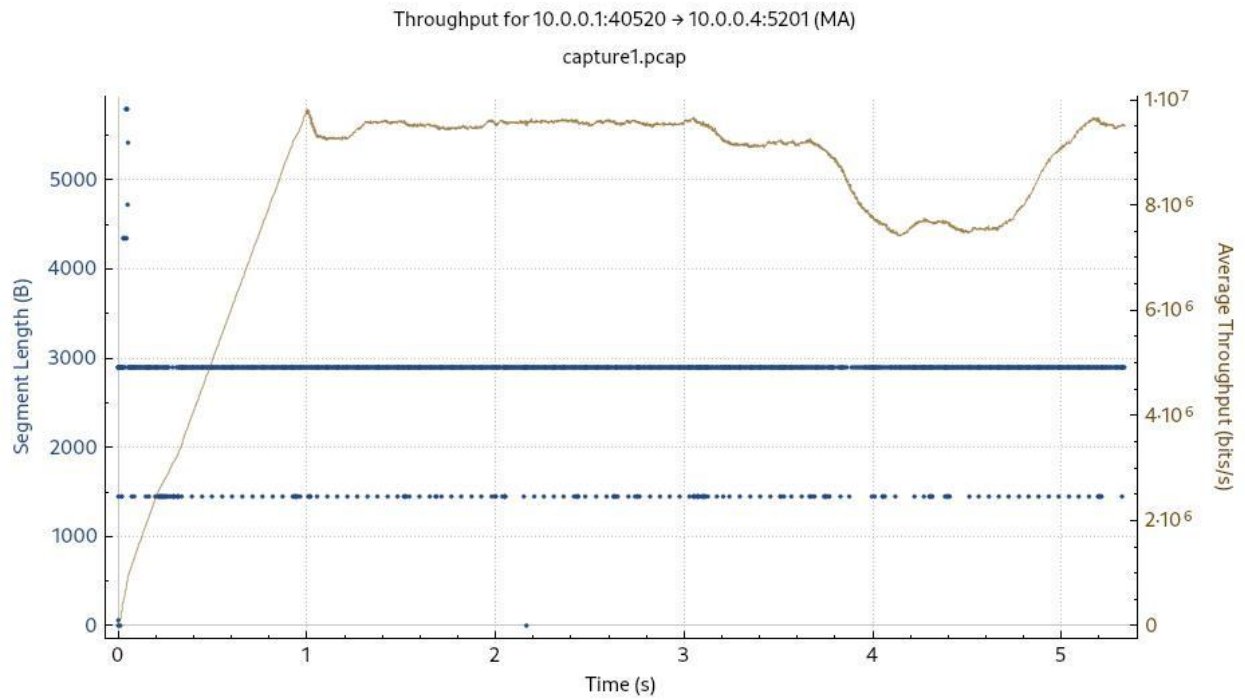


Fig. Throughput when link loss was 1%

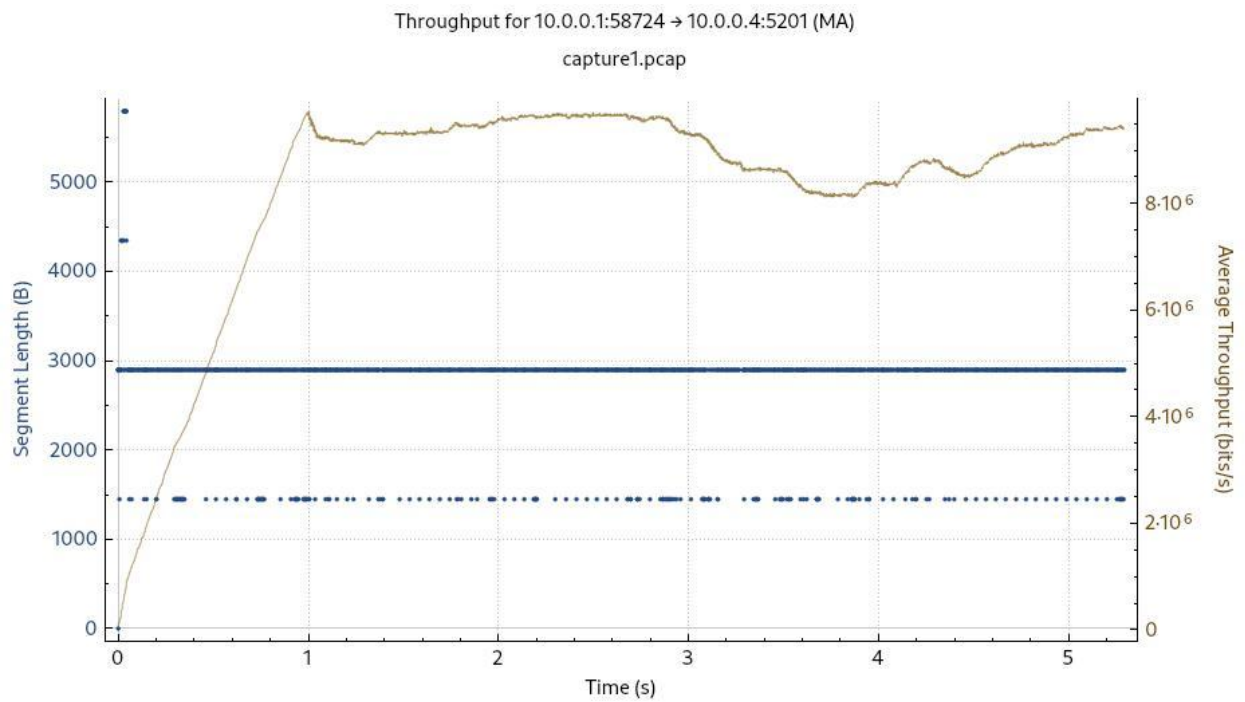


Fig. Throughput when link loss was 2%

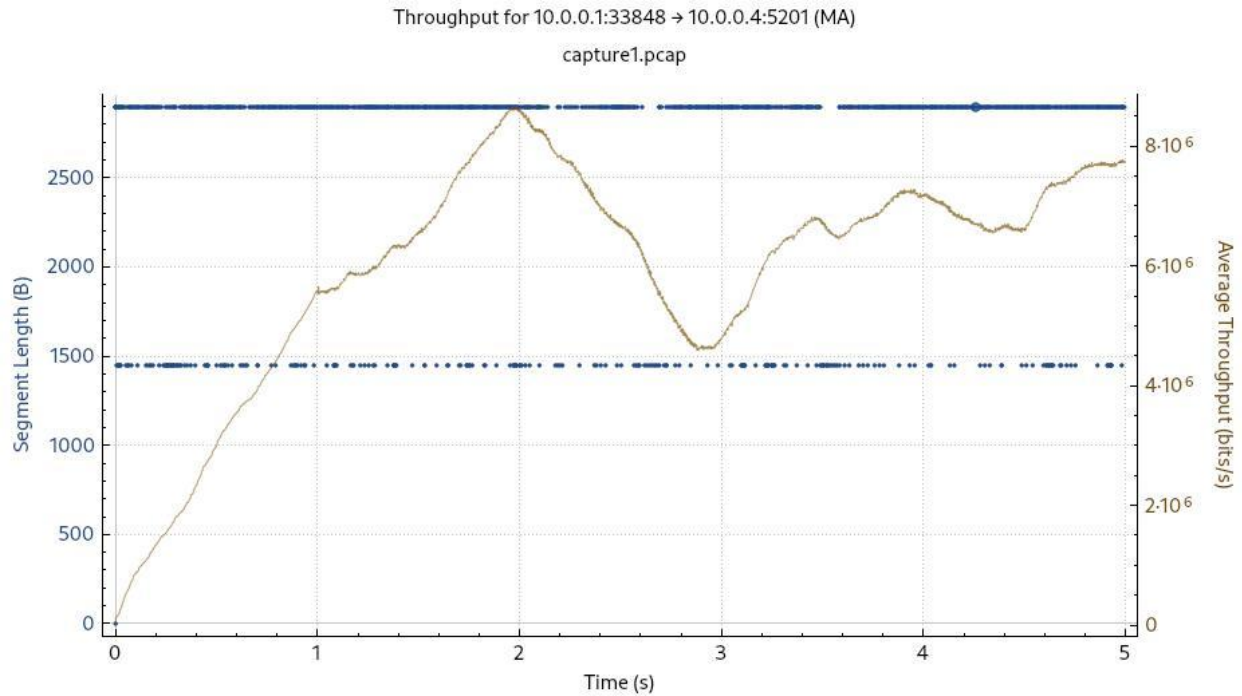


Fig. Throughput when link loss was 3%

Similar general trends have been observed again. The advantages of BBR are presumably not visible here since there are very few links in the paths that the packets are taking, hence the true advantages of BBR are not being utilized. Moreover, the bandwidth of the bottleneck link is also 10 MBPS, which is in contrast to the situations in which the BBR algorithm is designed to work (in modern scenarios with high link bandwidths).

Vegas

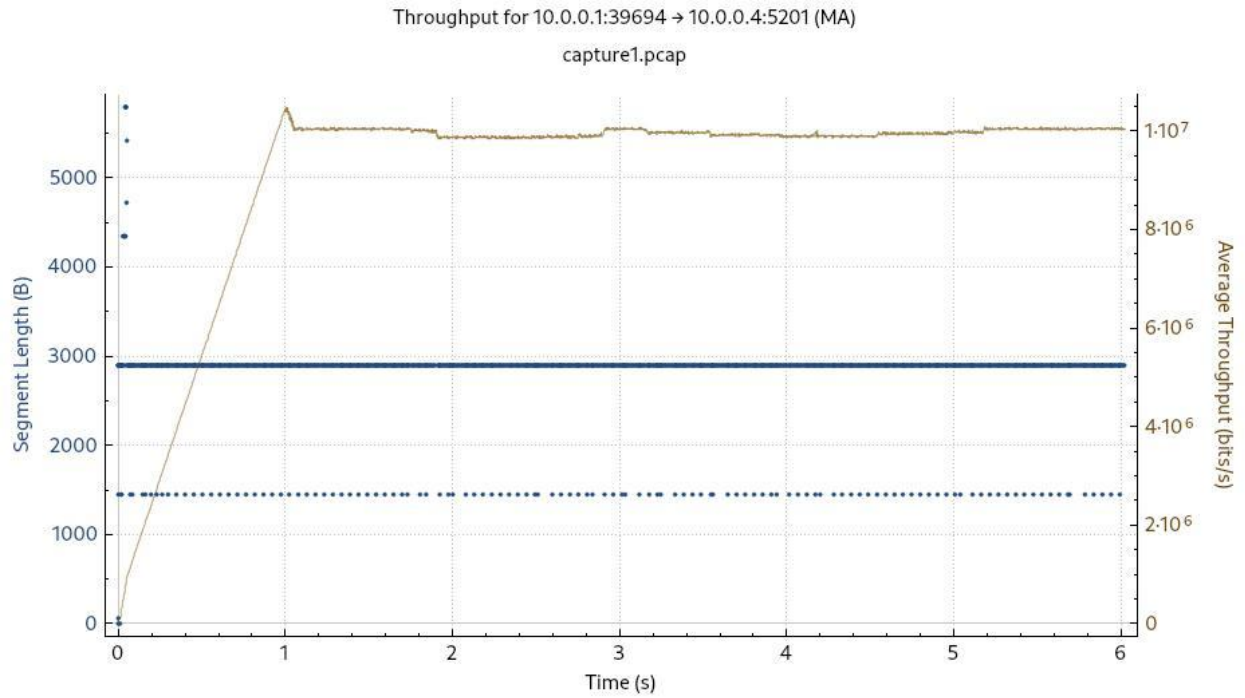


Fig. Throughput when link loss was 0%

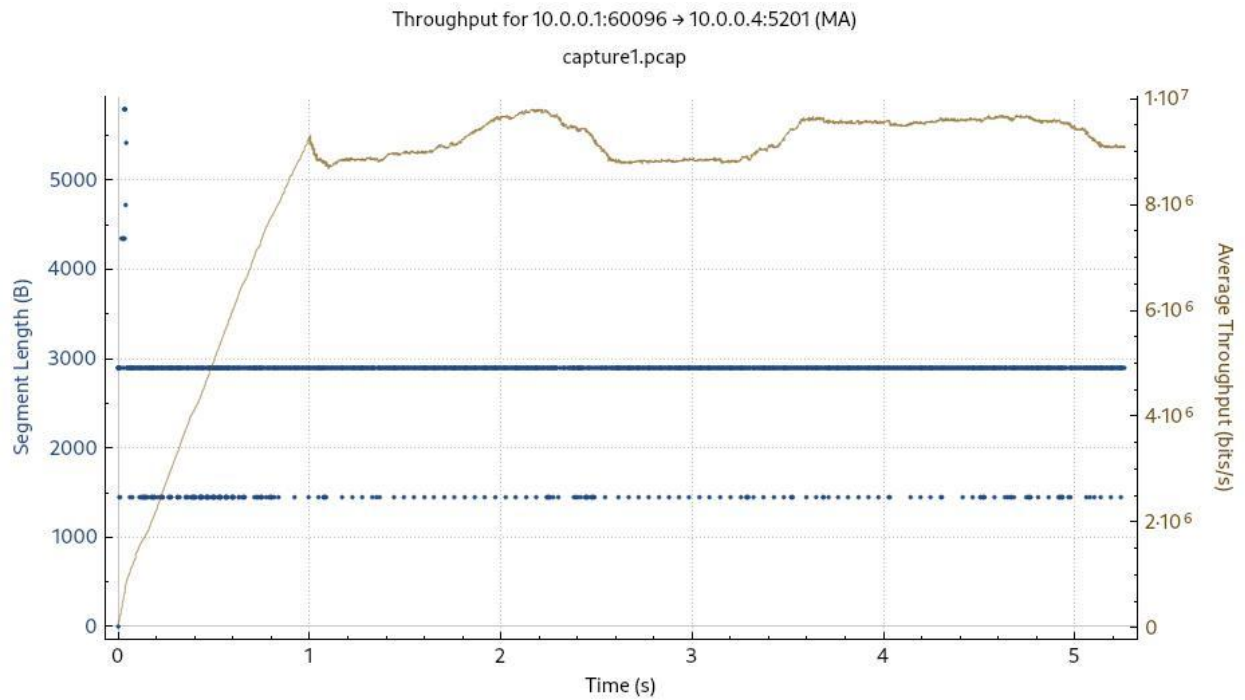


Fig. Throughput when link loss was 1%

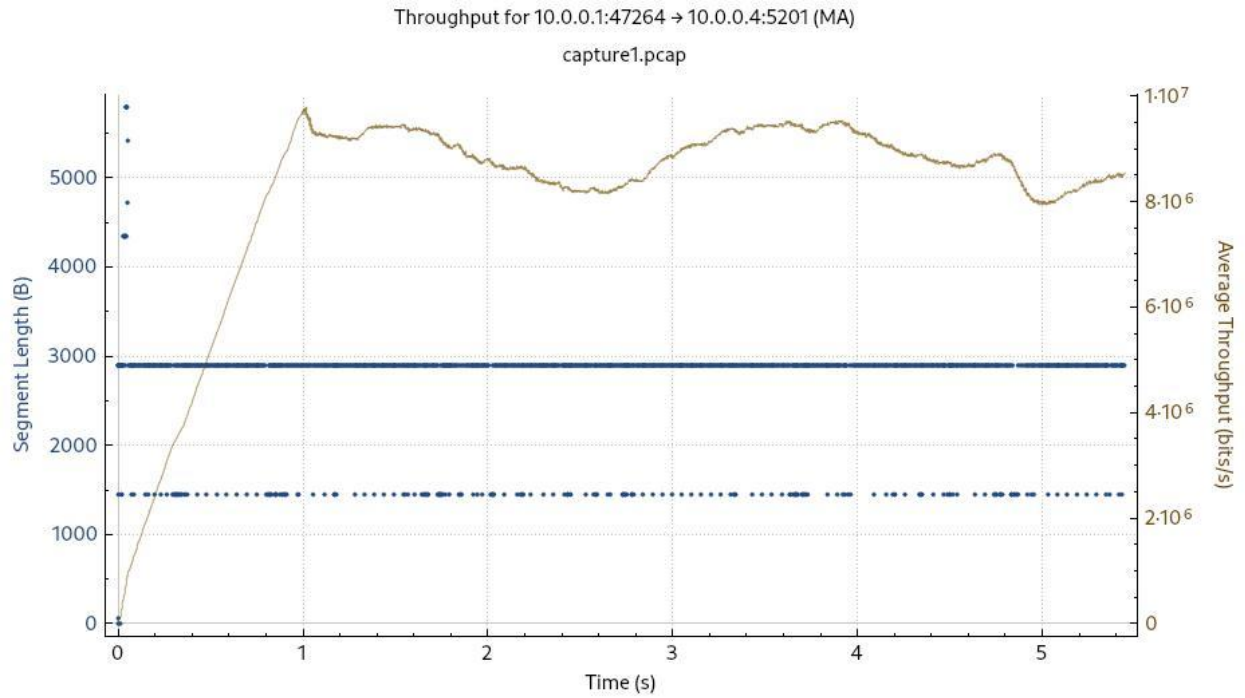


Fig. Throughput when link loss was 2%

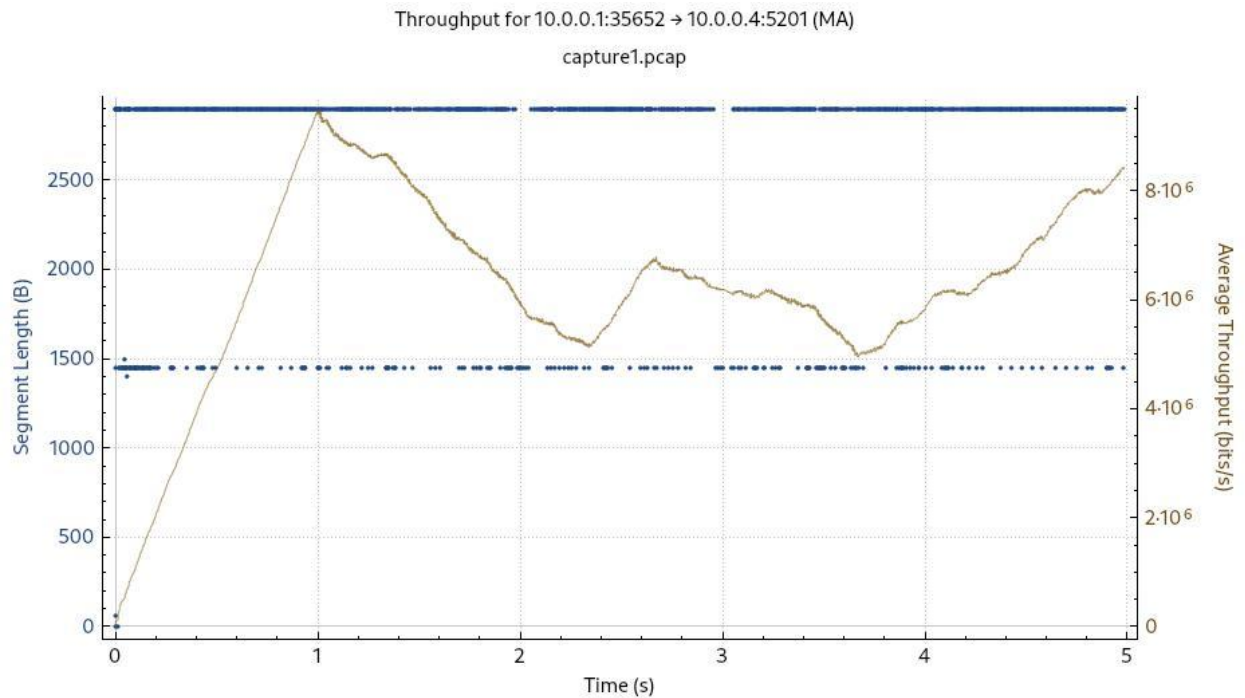


Fig. Throughput when link loss was 3%

The general trends have been retained. The distinct characteristics of vegas have not been exhibited here, owing to virtually constant RTT times, which is the main parameter tcp reno relies on. Hence the increase and decrease in the throughput occurs linearly.

Part c

Comment: All the graphs have been generated while keeping the link loss parameter as 0.

Cubic

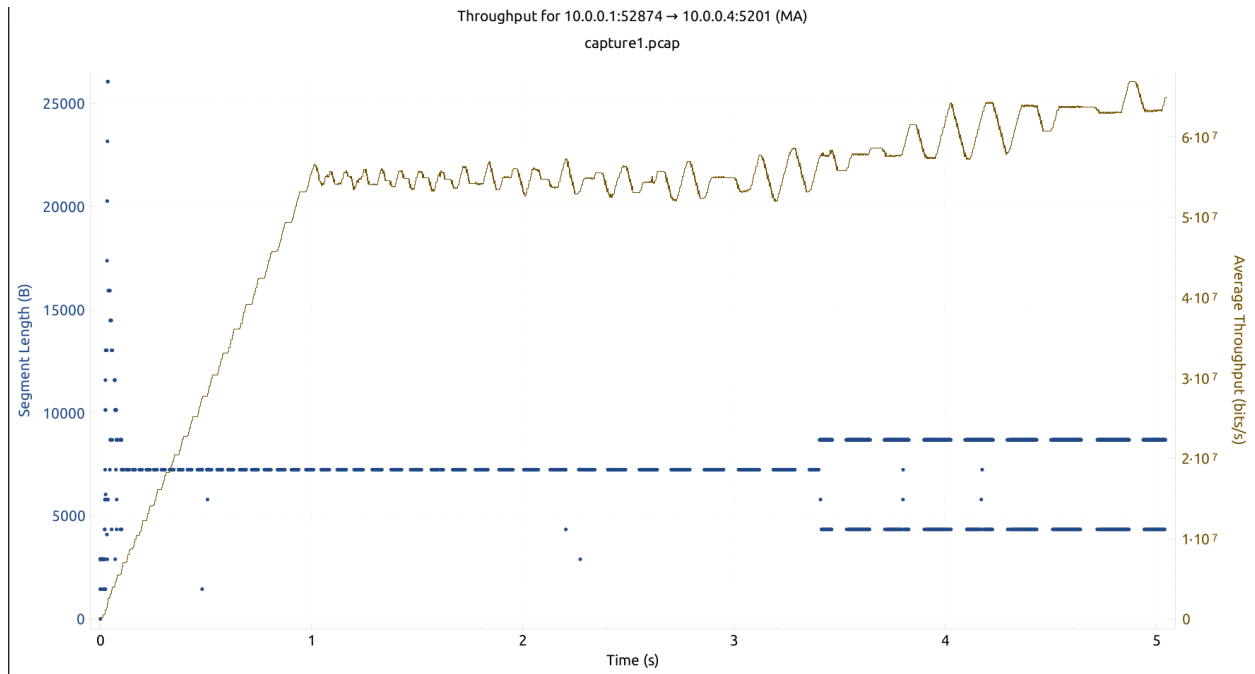


Fig. Throughput from host 1

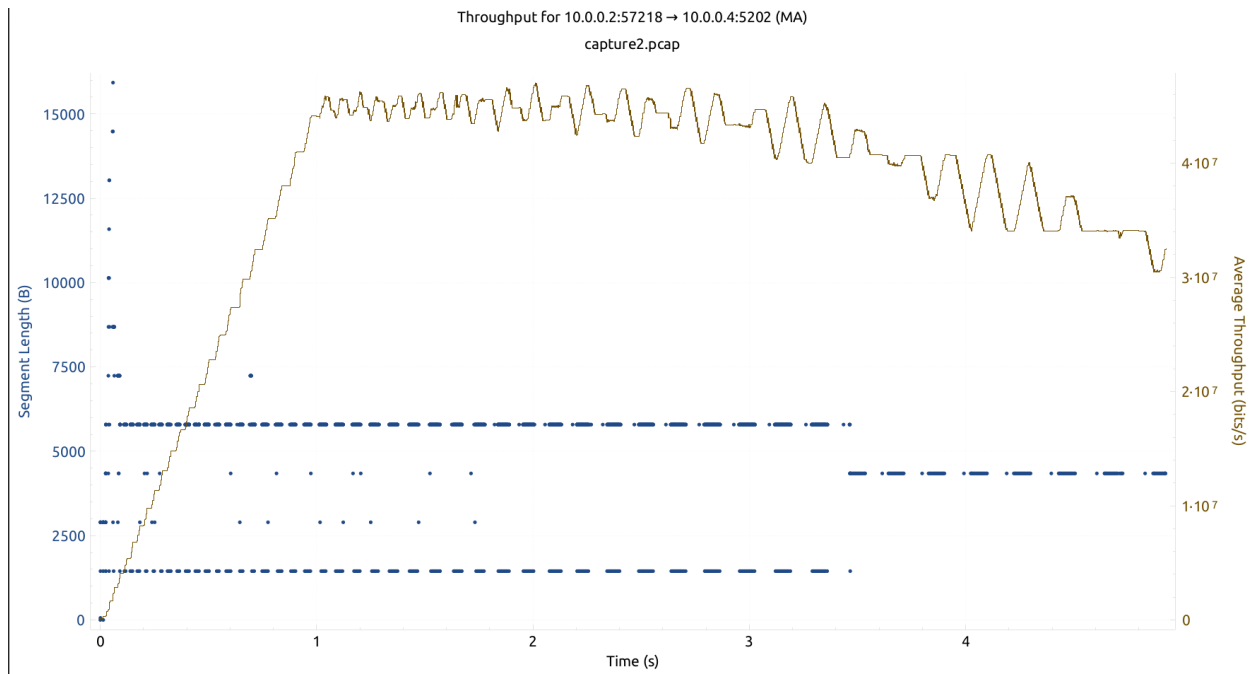


Fig. Throughput from host 2

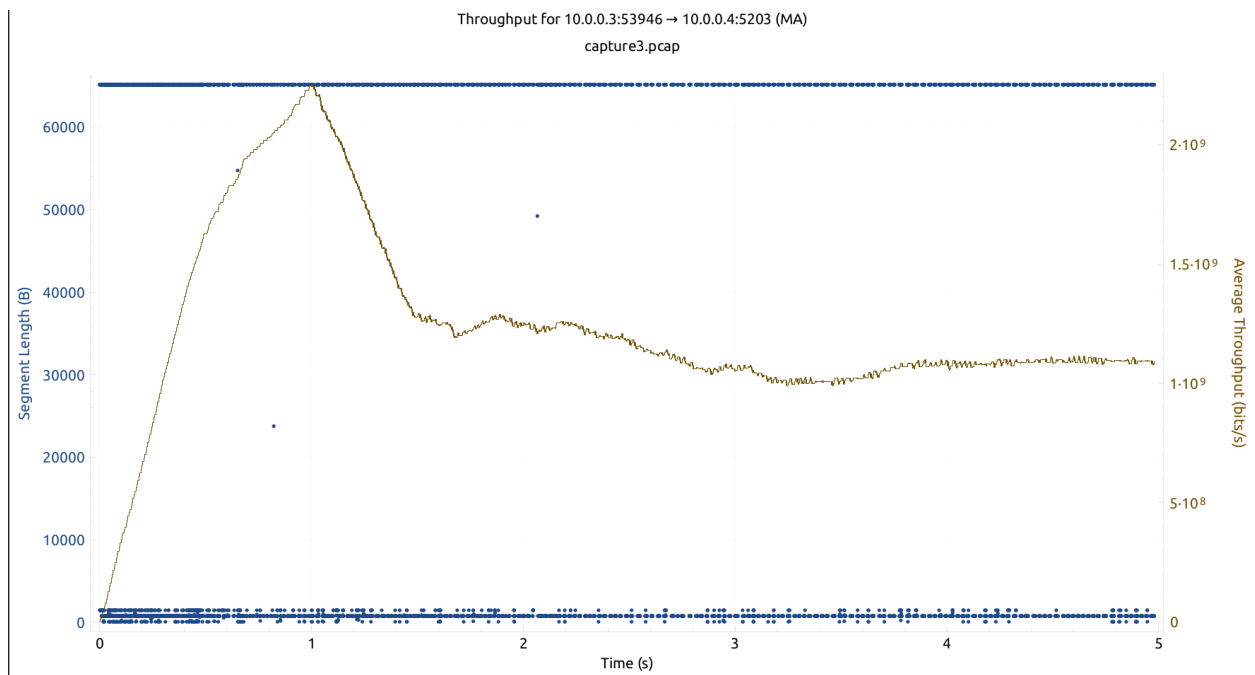


Fig. Throughput from host 3

The first observation is the fact that host 3 is able to attain much higher bandwidths than hosts 1 and 2 (in the order of 10^9 vs in the order of 10^7). This is an obvious result since host 3 can directly send packets to host 4 through switch 2, and hence it can bypass the link between switch 1 and switch 2, which is the main bottleneck in this network.

Up to a time of around 1 second, all of the hosts are able to increase their throughput without suffering any major bottlenecks. Afterwards, the network has become congested and then the congestion control algorithm comes into play.

Although the first host and second host are symmetric with respect to host 4, host 1 is still able to achieve higher throughput than host 2. This may be attributed to the fact that host 1 starts sending data earlier than host 2 since the iperf client in host 1 is set up earlier than in host 2.

The third host after the 1 second mark is forced to reduce its throughput, since presumably the switch too is experiencing congestion. Although the switch is software based, its capabilities are still constrained by the power of the system, and hence it too can experience congestion.

The cubic function is distinctly visible in many parts of graph 1 and 2, with the throughput increasing rapidly till an inflection point(which is the point at which packet loss previously occurred) after which it changes its concavity.

Reno

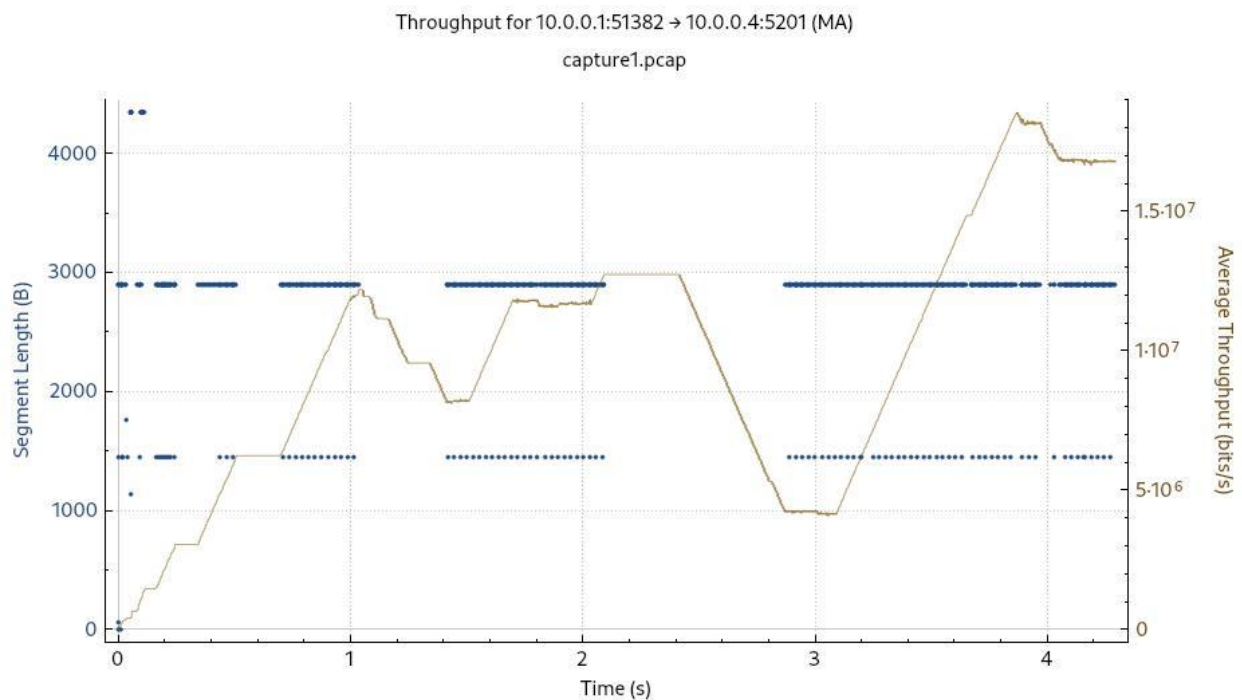


Fig. Throughput from host 1

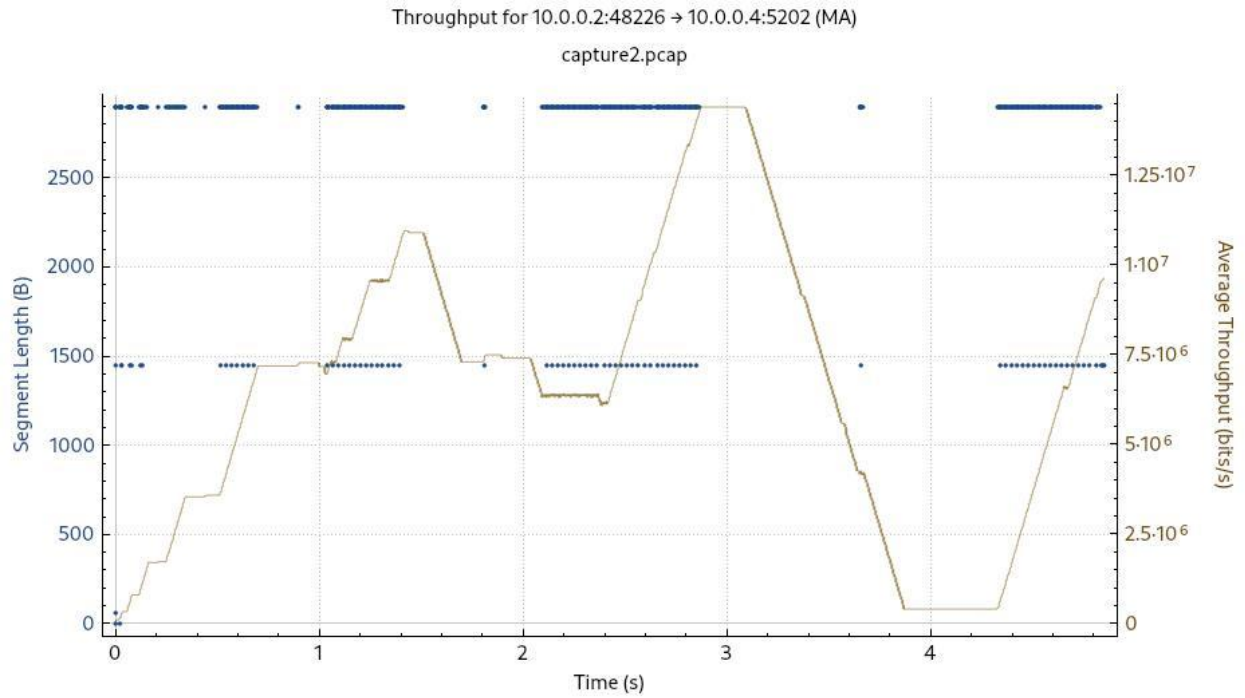


Fig. Throughput from host 2

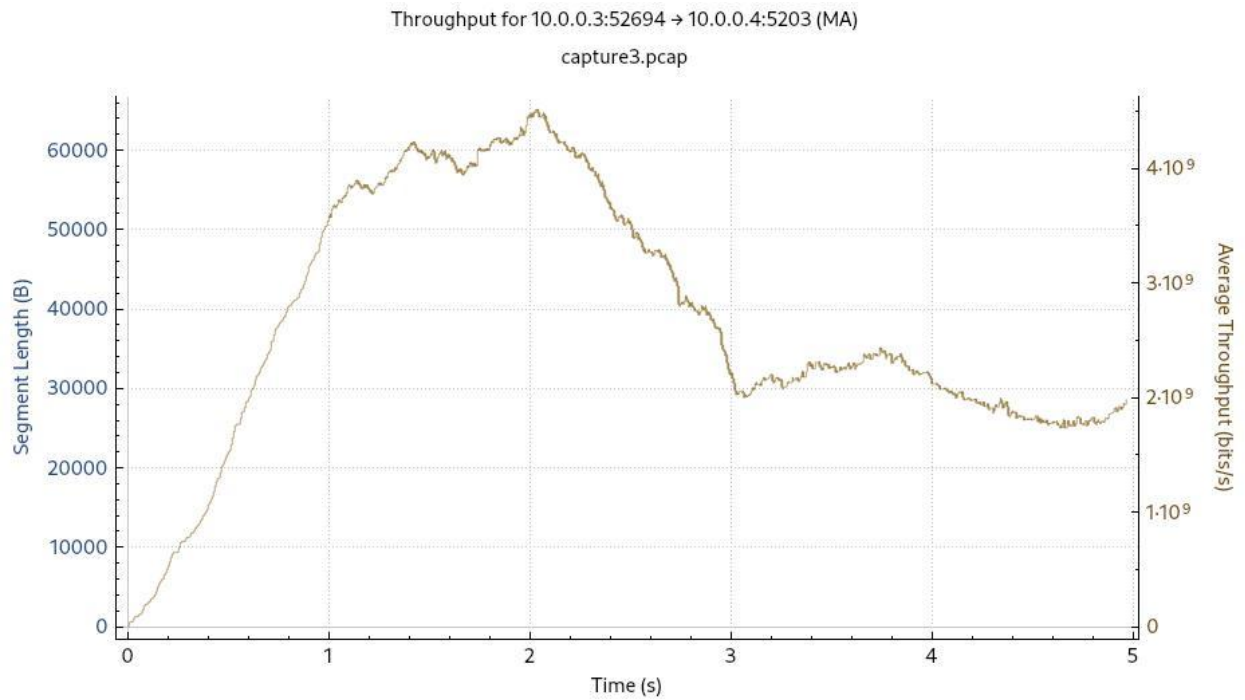


Fig. Throughput from host 3

The general trends from before are visible again, with host 3 attaining much higher throughput in comparison to host 1 and host 2.

The distinct characteristics of the TCP Reno algorithm are also clearly visible here, with linear increase and decrease modes. As has been mentioned before, the RTT times are nearly constant in the network. Hence the Reno algorithm makes changes only during congestion events and is not able to utilize RTT statistics to make corrections.

BBR

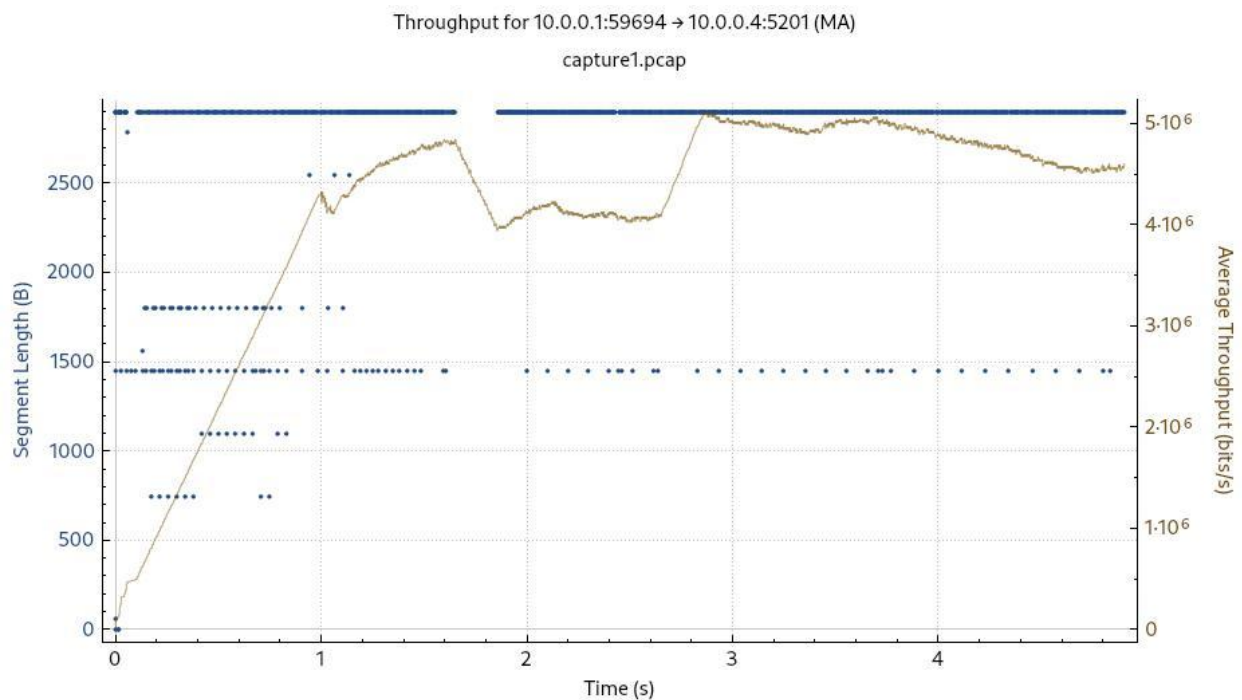


Fig. Throughput from host 1

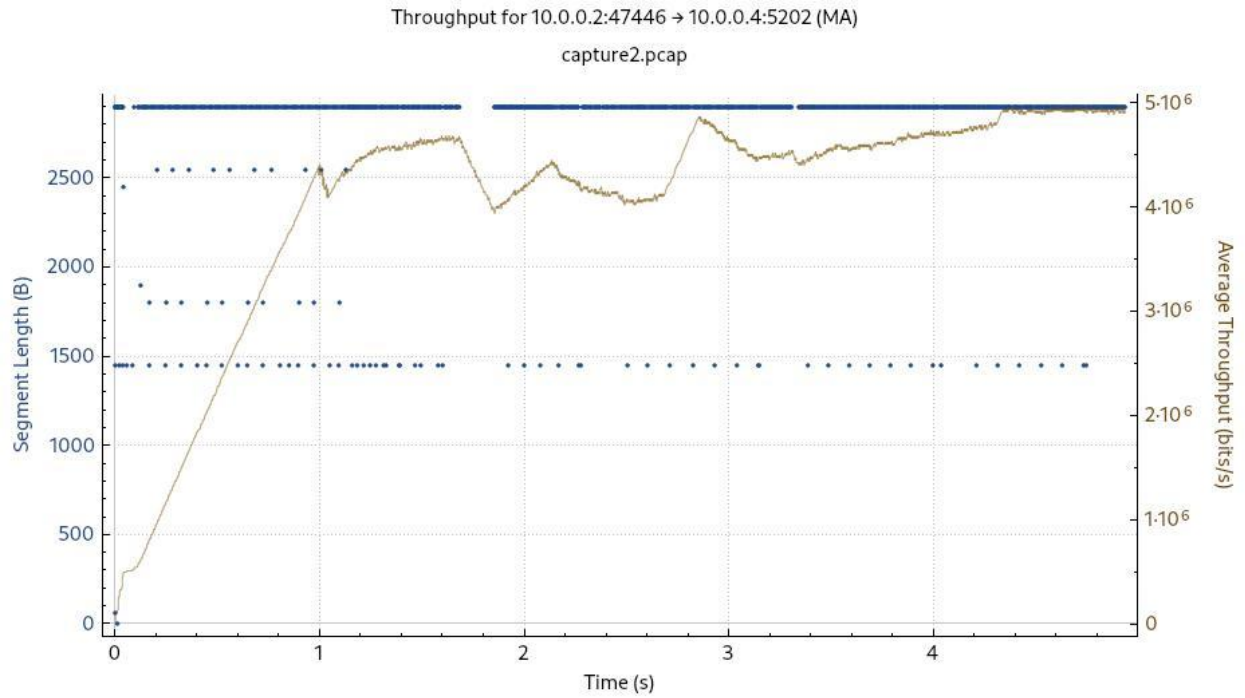


Fig. Throughput from host 2

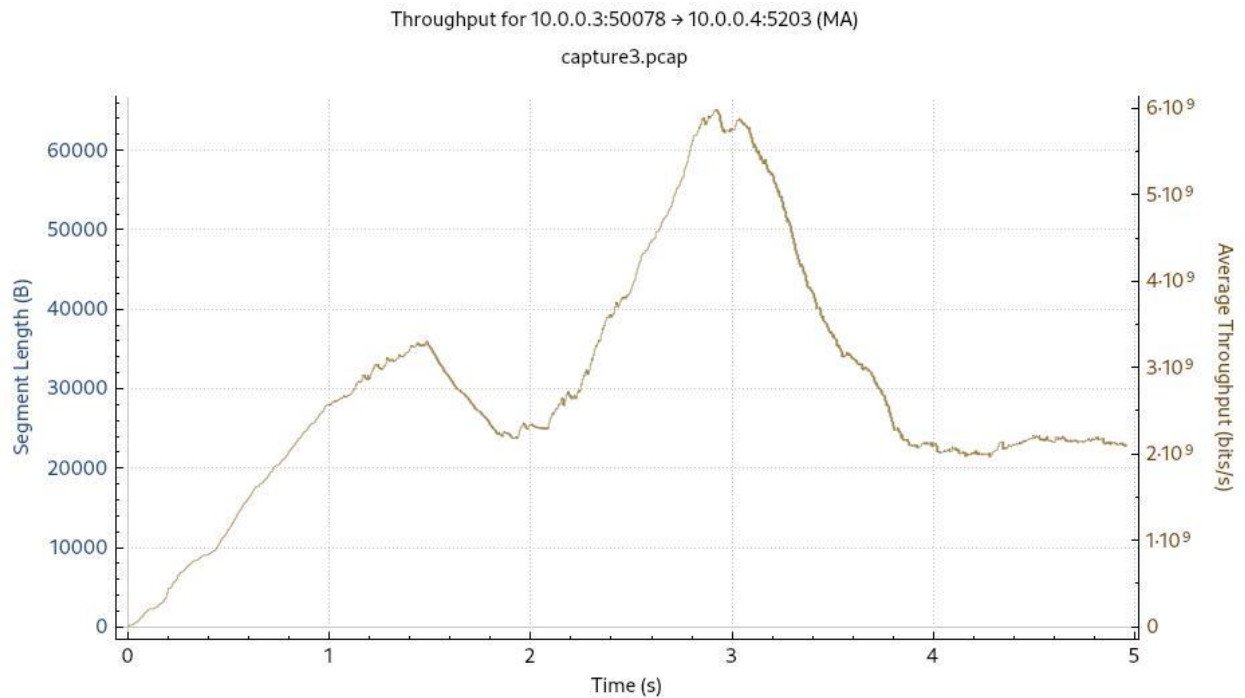


Fig. Throughput from host 3

The general trends from before have been retained. Host 3 again attains significantly higher throughput than the host 1 and host 2. For host 1 and host 2, the throughput is much more

stable, showing BBR's strengths when many hosts are sending data to a common link. BBR is a relatively complex algorithm in general, and hence not many general trends can be seen.

Host 3 has also shown higher variance in its throughput in comparison to the previous congestion control schemes. This may be attributed to BBR's higher sensitivity to high traffic.

Vegas

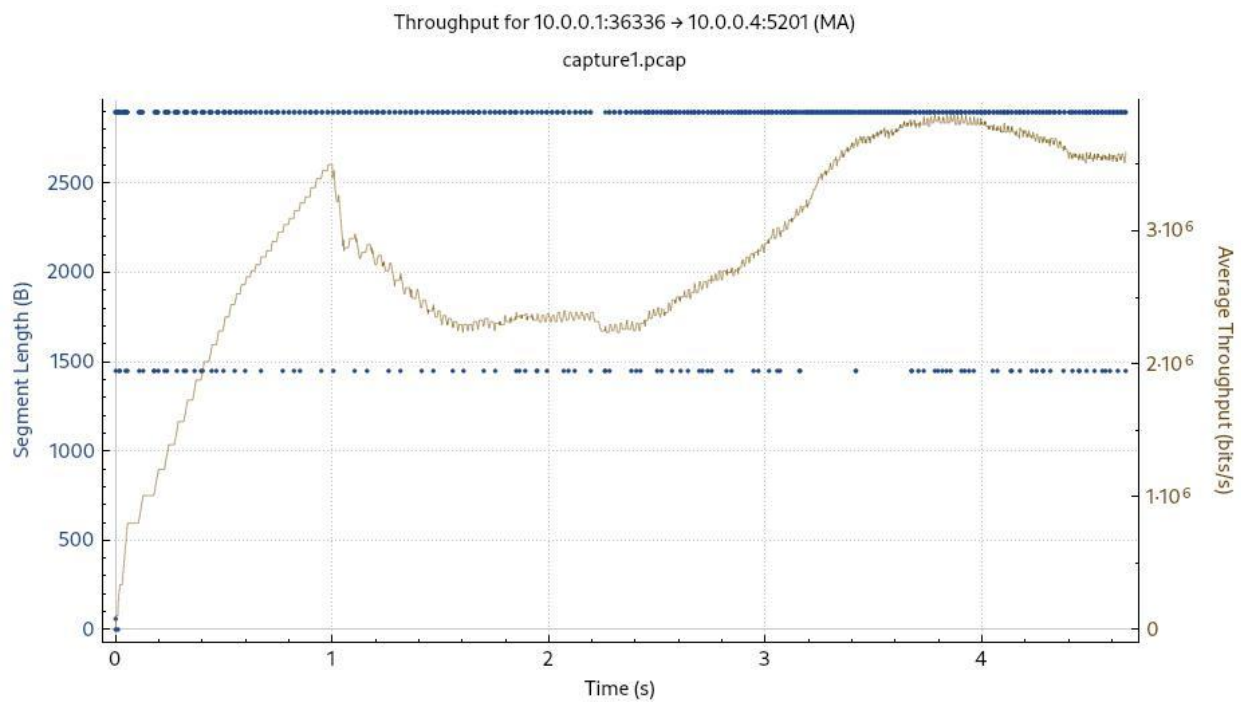


Fig. Throughput from host 1

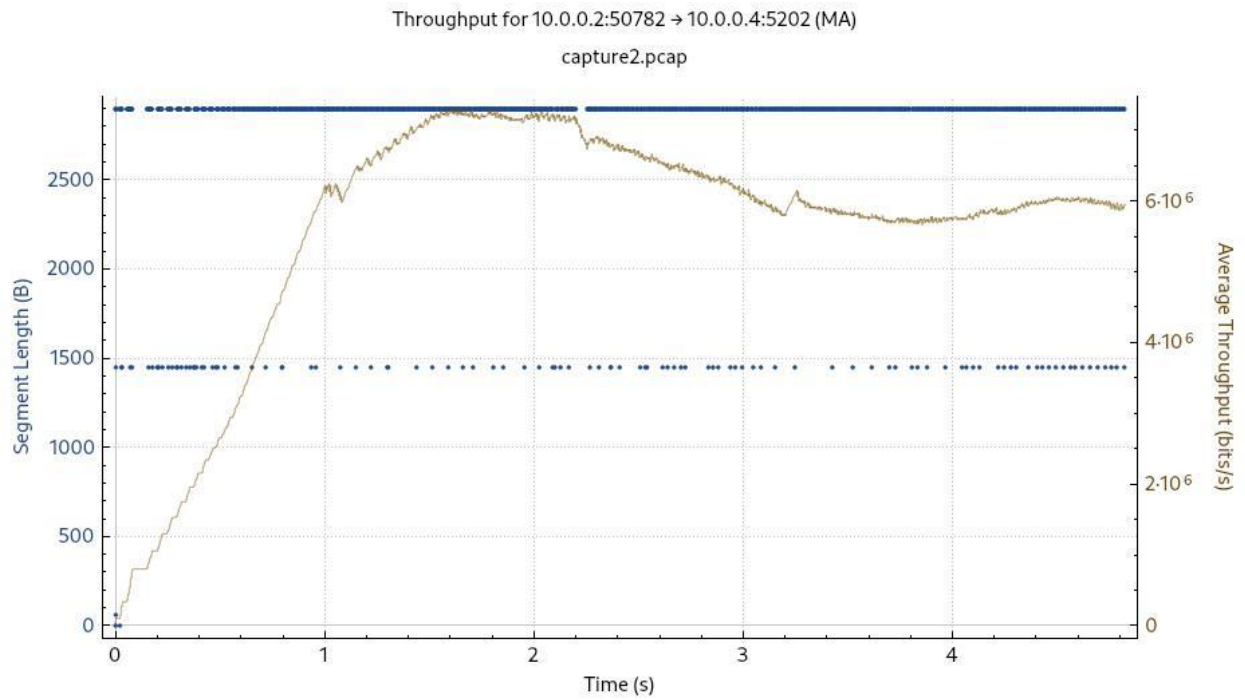


Fig. Throughput from host 2

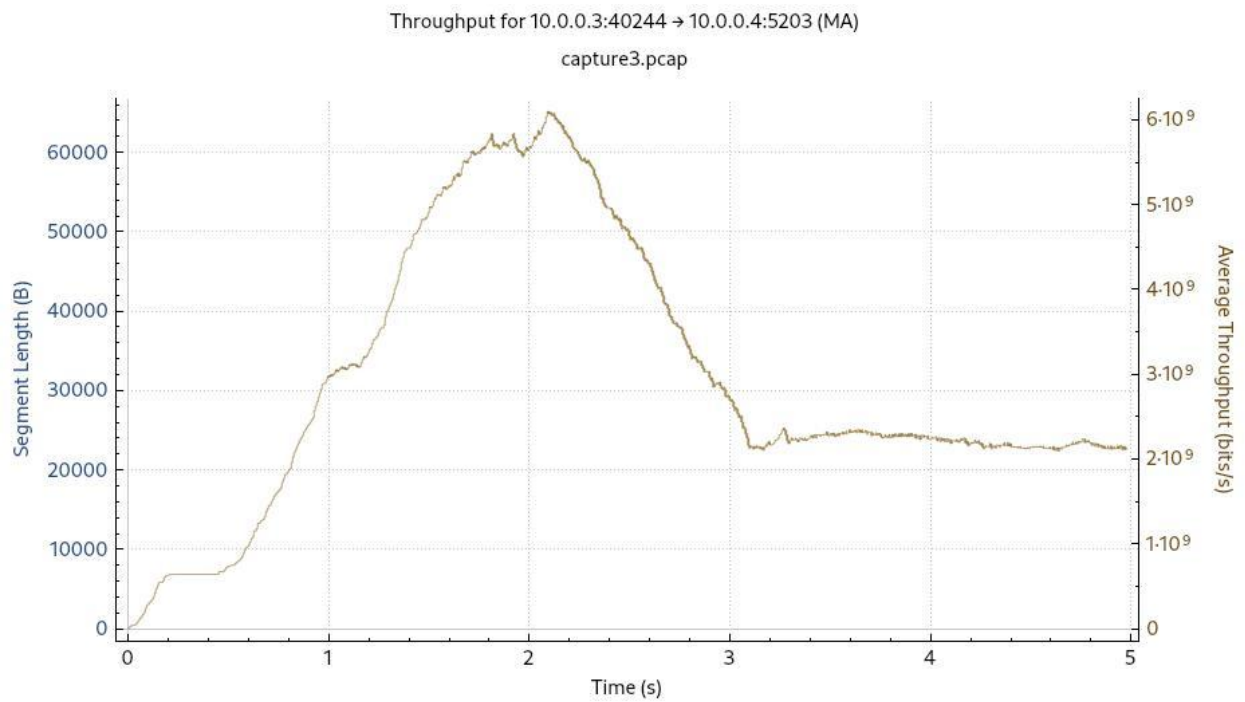


Fig. Throughput from host 3

Again, the general trend is observed again, with host 3 attaining much higher throughput.

The peculiarities of this congestion control algorithm are also visible here. In host 1 and host 2, the throughput is exhibiting period wave-like properties with high sensitivity to changing RTT times. These changing RTT times can be attributed to the hardware limitations of the software based switch.

Here too, host 1 and host 2 exhibit asymmetric behavior. This may again be attributed to the difference in their starting times.