CS 433 Computer Networks (2023-24)
Assignment-02

Sahil Das - 21110184
Yashraj J Deshmukh - 21110245

Part I: Implement the routing functionality in mininet.

a. Here's a brief description of the implementation of the required topology -

The first section of the code represents LinuxRouter class which is a custom Mininet node class designed to represent a router with IP forwarding capabilities..

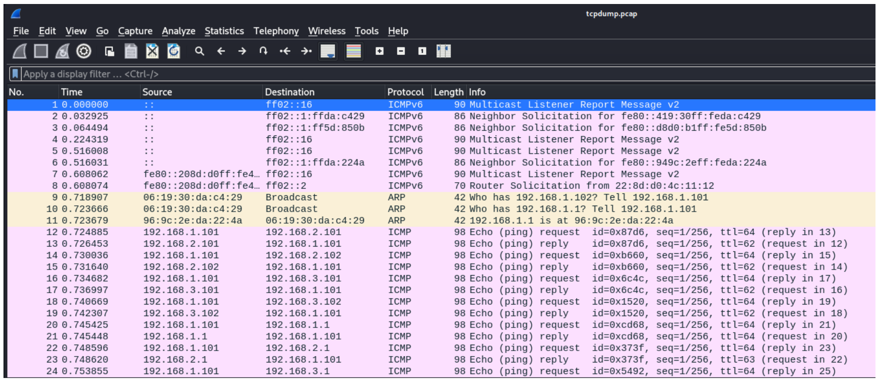
Next is NetworkTopo which is the class for custom topology. We implemented the routers using pairs of nodes and switches. Then to each switch of the router we connected two hosts, and then we proceeded to connect the nodes of the routers with each other. And, as a result, three subnet were created.

Now, to establish connection between hosts of different subnets, we went on to make the routing table and to do that we created three subnets that connects the routers.

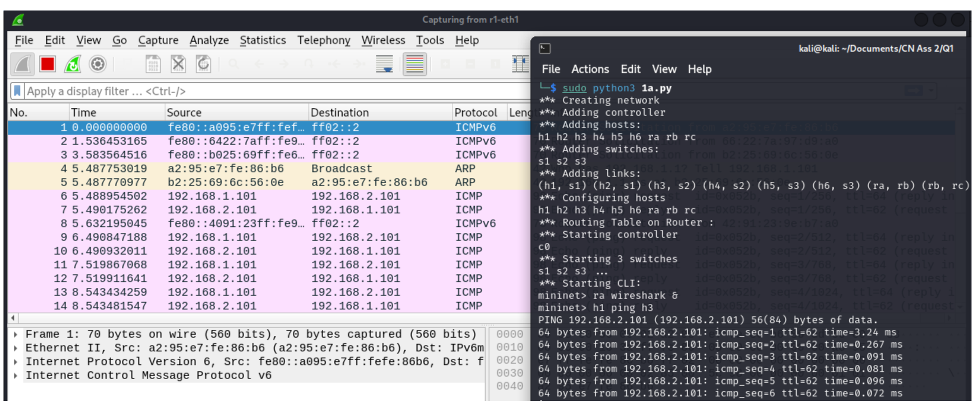
```
┌──(kali㉿kali)-[~/Documents/CN Ass 2]
└─$ sudo python3 Q1/1a.py
*** Creating network
*** Adding controller
*** Adding hosts:
h1 h2 h3 h4 h5 h6 ra rb rc
*** Adding switches:
s1 s2 s3
*** Adding links:
(h1, s1) (h2, s1) (h3, s2) (h4, s2) (h5, s3) (h6, s3) (ra, rb) (rb, rc) (rc, ra) (s1, ra) (s2, rb) (s3, rc)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 ra rb rc
*** Routing Table on Router :
*** Starting controller
c0
*** Starting 3 switches
s1 s2 s3 ...
*** Starting CLI:
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)
mininet>
```

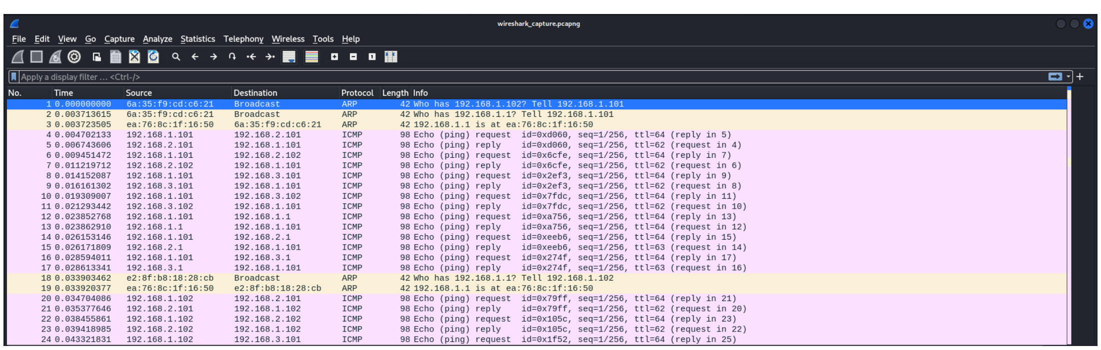
The results of pingall command clearly show that all hosts are inter-reachable. The "0% dropped" also reaffirms this conclusion.

b. The python program 1b.py is similar to the previous one in implementation, the only difference being that it automatically sets up a tcpdump for router ra and interface r1-eth1 following it up with a pingall. The packets transferred are stored in the tcpdump.pcap. Note that the packets corresponding to ICMP protocol are the ones involved during pingall (this can be verified by looking at their IPs). We see that there is no packet failure for this router cementing our conclusion from part a.



Alternatively, if we want to monitor the packets live you can instead run the python script 1a.py and in the mininet CLI type the command “r_i wireshark &” and choose the interface you wish to observe. The *&* here ensure that the wireshark runs in the background so that we can continue in our CLI independent of it. Now you can run any command on mn and observe the packets passed through the hook. In the submission, wireshark_capture.pcap stores the session packets of router *ra* with interface *r1-eth1* when we do a pingall command.





c. In this part we vary the route between two subnets, corresponding to routers *ra* and *rc*, in which their requests / packets are passed via router *rb* first. We incorporate this by changing the path definitions for both routes.

This is done by modifying the routing entries for *ra* and *rc.* In *ra,* we change the path to reach *rc* by passing it via the same link and interface as that between *ra* and *rb.* Similarly, we need to do this modification for router *rc* as well, in which the route to *ra* is updated to be same as the one for *rb.*

```
"Node: h1"

┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2]
└─# ping -c 4 192.168.3.102
PING 192.168.3.102 (192.168.3.102) 56(84) bytes of data.
64 bytes from 192.168.3.102: icmp_seq=1 ttl=62 time=4.25 ms
64 bytes from 192.168.3.102: icmp_seq=2 ttl=62 time=0.643 ms
64 bytes from 192.168.3.102: icmp_seq=3 ttl=62 time=0.116 ms
64 bytes from 192.168.3.102: icmp_seq=4 ttl=62 time=0.106 ms

--- 192.168.3.102 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3059ms
rtt min/avg/max/mdev = 0.106/1.278/4.250/1.729 ms

┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2]
└─# traceroute 192.168.3.102
traceroute to 192.168.3.102 (192.168.3.102), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  8.037 ms  6.727 ms  6.574 ms
 2  192.100.2.3 (192.100.2.3)  8.032 ms  8.685 ms  9.527 ms
 3  192.168.3.102 (192.168.3.102)  15.760 ms  16.129 ms  16.152 ms
```

ping results on original path

On tracing the route from h1 to h6, we observe that it hops from 192.168.1.101(h1) to 192.168.1.1(r1) to 192.100.2.3(r1->r3, intfName1='r3-eth3') to finally 192.168.3.102(h6).

```
*** Starting CLI:
mininet> h1 traceroute 192.168.3.102
traceroute to 192.168.3.102 (192.168.3.102), 30 hops max, 60 byte packets
 1  192.168.1.1 (192.168.1.1)  2.414 ms  0.926 ms  5.020 ms
 2  192.100.0.2 (192.100.0.2)  5.010 ms  5.014 ms  5.018 ms
 3  192.100.1.3 (192.100.1.3)  5.020 ms  5.024 ms  5.033 ms
 4  192.168.3.102 (192.168.3.102)  7.467 ms  7.479 ms  7.486 ms
mininet> █
```

ping results on new path

On tracing the modified route from h1 to h6, we observe that it hops from 192.168.1.101(h1) to 192.168.1.1(r1) like previously. But from here it hops to 192.100.0.2(r1->r2, intfName2='r2-eth2') and then to 192.100.1.3(r2->r3, intfName2='r3-eth2') and then the packet finally reaches 192.168.3.102(h6).

ping rtt comparison:

```
mininet> h1 ping h6 -c 4
PING 192.168.3.102 (192.168.3.102) 56(84) bytes of data.
64 bytes from 192.168.3.102: icmp_seq=1 ttl=62 time=0.051 ms
64 bytes from 192.168.3.102: icmp_seq=2 ttl=62 time=0.063 ms
64 bytes from 192.168.3.102: icmp_seq=3 ttl=62 time=0.082 ms
64 bytes from 192.168.3.102: icmp_seq=4 ttl=62 time=0.062 ms

--- 192.168.3.102 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3082ms
rtt min/avg/max/mdev = 0.051/0.064/0.082/0.011 ms
```

Default Route

```
mininet> h1 ping h6 -c 4
PING 192.168.3.102 (192.168.3.102) 56(84) bytes of data.
64 bytes from 192.168.3.102: icmp_seq=1 ttl=61 time=0.535 ms
64 bytes from 192.168.3.102: icmp_seq=2 ttl=61 time=0.116 ms
64 bytes from 192.168.3.102: icmp_seq=3 ttl=61 time=0.122 ms
64 bytes from 192.168.3.102: icmp_seq=4 ttl=61 time=0.211 ms

--- 192.168.3.102 ping statistics ---
4 packets transmitted, 4 received, 0% packet loss, time 3075ms
rtt min/avg/max/mdev = 0.116/0.246/0.535/0.171 ms
```

Detour

In the route taken by packets to go from h1 to h6, it now has to pass through 3 routers, whereas in the default route it only passes through two, ra and rc. So, the path length, excluding switch-host, is about 2 units for the detour as compared to the 1 unit for normal route. In addition to this, all the links on the network have some delay (except in the internal working of routers) to replicate real-life latency. Since the detour passes through more links, it will naturally have more propagation delay.

This all is reflected in the RTT for both cases, as the one for detour is roughly double that of the default path.

iperf comparison:

```
"Node: h6"
┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2/Q1]
└─# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.3.102 port 5001 connected with 192.168.1.101 port 56772 (icw
nd/mss/irtt=14/1448/3124)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0018 sec  34.8 GBytes  29.9 Gbits/sec
[  2] local 192.168.3.102 port 5001 connected with 192.168.2.101 port 35260 (icw
nd/mss/irtt=14/1448/4669)
[ ID] Interval       Transfer     Bandwidth
[  2] 0.0000-9.9965 sec  38.2 GBytes  32.8 Gbits/sec
```

```
"Node: h1"
┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2/Q1]
└─# iperf -c 192.168.3.102
------------------------------------------------------------
Client connecting to 192.168.3.102, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.1.101 port 56772 connected with 192.168.3.102 port 5001 (icw
nd/mss/irtt=14/1448/6731)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0145 sec  34.8 GBytes  29.9 Gbits/sec
```

```
"Node: h3"
┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2/Q1]
└─# iperf -c 192.168.3.102
------------------------------------------------------------
Client connecting to 192.168.3.102, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.2.101 port 35260 connected with 192.168.3.102 port 5001 (icw
nd/mss/irtt=14/1448/2964)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0089 sec  38.2 GBytes  32.7 Gbits/sec
```

Default Routing

```
"Node: h6"
┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2/Q1]
└─# iperf -s
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.3.102 port 5001 connected with 192.168.1.101 port 45526 (icw
nd/mss/irtt=14/1448/2209)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0003 sec  31.5 GBytes  27.0 Gbits/sec
```

```
"Node: h1"
┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2/Q1]
└─# iperf -c 192.168.3.102
------------------------------------------------------------
Client connecting to 192.168.3.102, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.1.101 port 45526 connected with 192.168.3.102 port 5001 (icw
nd/mss/irtt=14/1448/3785)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0060 sec  31.5 GBytes  27.0 Gbits/sec
```

```
"Node: h3"
┌──(root㉿kali)-[/home/kali/Documents/CN Ass 2/Q1]
└─# iperf -c 192.168.3.102
------------------------------------------------------------
Client connecting to 192.168.3.102, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 192.168.2.101 port 39176 connected with 192.168.3.102 port 5001 (icw
nd/mss/irtt=14/1448/3263)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0073 sec  37.9 GBytes  32.5 Gbits/sec
```

Modified Routing

On comparing the iperf results of the two routings, the throughput for h6←→h3 is nearly the same for both cases, but for h6←→h1 we observe a dip in the value for the later case. This may be credited to the fact that the path between h3 and h6 remains the same but for h1 and h6 it is altered.

d. The routing table for each “router” is obtained by running the following command for each of the routers - mininet.log.info("For router r_i:", net['r_i'].cmd('route'))
An alternative to this approach is to run *r_i route* in the mininet CLI for each router.

You can view the output given below by un-commenting this section in both codes. Note that in part_c the output will be printed when you exit mininet.

For part a:

```
For router ra: Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.100.0.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth2
192.100.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth3
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth1
192.168.2.0     192.100.0.2     255.255.255.0   UG    0      0        0 r1-eth2
192.168.3.0     192.100.2.3     255.255.255.0   UG    0      0        0 r1-eth3

For router rb: Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.100.0.0     0.0.0.0         255.255.255.0   U     0      0        0 r2-eth2
192.100.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r2-eth3
192.168.1.0     192.100.0.1     255.255.255.0   UG    0      0        0 r2-eth2
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r2-eth1
192.168.3.0     192.100.1.3     255.255.255.0   UG    0      0        0 r2-eth3

For router rc: Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.100.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth2
192.100.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth3
192.168.1.0     192.100.2.1     255.255.255.0   UG    0      0        0 r3-eth3
192.168.2.0     192.100.1.2     255.255.255.0   UG    0      0        0 r3-eth2
192.168.3.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth1
```

For part c:

```
For router ra: Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.100.0.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth2
192.100.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth3
192.168.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r1-eth1
192.168.2.0     192.100.0.2     255.255.255.0   UG    0      0        0 r1-eth2
192.168.3.0     192.100.0.2     255.255.255.0   UG    0      0        0 r1-eth2

For router rb: Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.100.0.0     0.0.0.0         255.255.255.0   U     0      0        0 r2-eth2
192.100.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r2-eth3
192.168.1.0     192.100.0.1     255.255.255.0   UG    0      0        0 r2-eth2
192.168.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r2-eth1
192.168.3.0     192.100.1.3     255.255.255.0   UG    0      0        0 r2-eth3

For router rc: Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
192.100.1.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth2
192.100.2.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth3
192.168.1.0     192.100.1.2     255.255.255.0   UG    0      0        0 r3-eth2
192.168.2.0     192.100.1.2     255.255.255.0   UG    0      0        0 r3-eth2
192.168.3.0     0.0.0.0         255.255.255.0   U     0      0        0 r3-eth1
```

Part II: Throughput for different congestion control schemes.

A. Implementation details:

The first part of the implementation defines and parses the command line arguments that are given by the user such as --config= , --loss= , and --congestion=. By default configuration is set as ‘b’, loss equals 0, and congestion algorithm as Cubic.

The next part is about building the custom topology that was given by adding the hosts, switches and connecting them with the help of addLink. Also, to observe congestion the bandwidth between s1-s2 is set to 10Mbits/sec.

The next part consists of starting the server in h4 using iperf command and running it in the background. Then the configuration is matched, and the host are selected respectively. Once it is done a for loop is used to iterate over the host in order to make each one of those a client by using the iperf command and setting the time limit as 5 sec, interval length as 1sec, and congestion control algorithm as decided by the user.

Then the output is stored in a string as well as in a pcap and text file but those are optional. The pcap file is later used to create the graph of throughput vs time using wireshark. At the end mininet is terminated.

B.

```
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.88 MBytes  15.7 Mbits/sec
[  1] 1.0000-2.0000 sec   825 KBytes  6.76 Mbits/sec
[  1] 2.0000-3.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 3.0000-4.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 4.0000-5.0000 sec   881 KBytes  7.21 Mbits/sec
```

Figure - Throughput vs time when congestion algo is set as Reno

Initially, throughput was a large value then it quickly dropped to around 6 Mbits/sec then it started to increase uptil 10.5 with unequal steps indicating doubling the packets every other second. Lastly, it again dropped because the network become congested. All this indicates that it should be TCP Reno congestion algorithm.

```
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  3.57 MBytes  29.9 Mbits/sec
[  1] 1.0000-2.0000 sec   637 KBytes  5.22 Mbits/sec
[  1] 2.0000-3.0000 sec   826 KBytes  6.76 Mbits/sec
[  1] 3.0000-4.0000 sec  1.12 MBytes  9.40 Mbits/sec
[  1] 4.0000-5.0000 sec   954 KBytes  7.81 Mbits/sec
[  1] 5.0000-7.1281 sec   125 KBytes   481 Kbits/sec
[  1] 0.0000-7.1281 sec  7.17 MBytes  8.44 Mbits/sec
```

Figure - Throughput vs time when congestion algo is set as Cubic

As we can observe at first the throughput was large then at the next moment it went down(can be said the cwd was halved, as seen from the next data points). After that it started to increase little by little but the step size increase every second which is more then the previous case, then it hits its maximum and drops again. This gives us an indication that the congestion control algorithm that is used is TCP Cubic.

```
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.38 MBytes  11.5 Mbits/sec
[  1] 1.0000-2.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 2.0000-3.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 3.0000-4.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 4.0000-5.0000 sec   640 KBytes  5.24 Mbits/sec
[  1] 5.0000-5.8432 sec   128 KBytes  1.24 Mbits/sec
[  1] 0.0000-5.8432 sec  5.00 MBytes  7.18 Mbits/sec
```

Figure - Throughput vs time when congestion algo is set as Vegas

In the above figure we can observe that the throughput is decreased for the first 3 seconds, then it increased again and the same trend continues. Which is the property of TCP Vegas congestion control algorithm.

```
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.63 MBytes  13.6 Mbits/sec
[  1] 1.0000-2.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 2.0000-3.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 3.0000-4.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 4.0000-5.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 5.0000-5.7831 sec   128 KBytes  1.34 Mbits/sec
[  1] 0.0000-5.7831 sec  6.13 MBytes  8.88 Mbits/sec
```

Figure - Throughput vs time when congestion algo is set as BBR.

C.

```
Client connecting to 10.0.1.4, TCP port 5001
TCP congestion control set to reno
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.1.1 port 38588 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/5470)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec   945 KBytes  7.74 Mbits/sec
[  1] 1.0000-2.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 2.0000-3.0000 sec   768 KBytes  6.29 Mbits/sec
[  1] 3.0000-4.0000 sec   768 KBytes  6.29 Mbits/sec
[  1] 4.0000-5.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 5.0000-5.6002 sec   128 KBytes  1.75 Mbits/sec
[  1] 0.0000-5.6002 sec  4.30 MBytes  6.44 Mbits/sec
```

```
[  1] local 10.0.1.2 port 56300 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/2572)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.75 MBytes  14.7 Mbits/sec
[  1] 1.0000-2.0000 sec   809 KBytes  6.62 Mbits/sec
[  1] 2.0000-3.0000 sec  1.38 MBytes  11.5 Mbits/sec
[  1] 3.0000-4.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 4.0000-5.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 5.0000-6.4508 sec   128 KBytes   723 Kbits/sec
[  1] 0.0000-6.4508 sec  5.79 MBytes  7.53 Mbits/sec
```

```
[  1] local 10.0.1.3 port 35270 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/3978)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  5.85 GBytes  50.3 Gbits/sec
[  1] 1.0000-2.0000 sec  6.26 GBytes  53.7 Gbits/sec
[  1] 2.0000-3.0000 sec  6.28 GBytes  53.9 Gbits/sec
[  1] 3.0000-4.0000 sec  6.33 GBytes  54.4 Gbits/sec
[  1] 4.0000-5.0000 sec  6.17 GBytes  53.0 Gbits/sec
[  1] 0.0000-5.0100 sec  30.9 GBytes  53.0 Gbits/sec
```

Figure - Throughput vs time when congestion algo is set as Reno for h1, h2 and h3 respectively.

One of the notable things to observe is that when the 3 clients are simultaneously using the link. Two of them have throughput around Mbits/sec and the h3 host is having around 50 Gbits/sec. It can be inferred from the fact that since only h1 and h2 has to use the link between the switch which has a bandwidth of 10Mbits/sec and in this case it is the bottleneck hence, the values differed. Also, it can be seen that one of the h1 or h2 has less throughput than the other one at different time instants which goes according to the theory when more then one host use the same connection, i.e., at time of congestion one remains as it is and the other goes down. Also, by observing the values of throughput obtained by h2 at a time it was at around 6Mbits/sec and in the next moment it went to around 12Mbits/sec, and after reaching a certain value it increased slowly. This indicates that TCP Reno was used.

```
[  1] local 10.0.1.1 port 43302 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/5307)
tcpdump: listening on h1-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.63 MBytes  13.6 Mbits/sec
[  1] 1.0000-2.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 2.0000-3.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 3.0000-4.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 4.0000-5.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 5.0000-5.4659 sec   128 KBytes  2.25 Mbits/sec
[  1] 0.0000-5.4659 sec  5.88 MBytes  9.02 Mbits/sec
```

```
[  1] local 10.0.1.2 port 36582 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/2631)
tcpdump: listening on h2-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.38 MBytes  11.5 Mbits/sec
[  1] 1.0000-2.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 2.0000-3.0000 sec   896 KBytes  7.34 Mbits/sec
[  1] 3.0000-4.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 4.0000-5.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 5.0000-5.4249 sec   128 KBytes  2.47 Mbits/sec
[  1] 0.0000-5.4249 sec  5.63 MBytes  8.70 Mbits/sec
```

```
[  1] local 10.0.1.3 port 56100 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/4604)
tcpdump: listening on h3-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  5.37 GBytes  46.1 Gbits/sec
[  1] 1.0000-2.0000 sec  5.83 GBytes  50.1 Gbits/sec
[  1] 2.0000-3.0000 sec  5.62 GBytes  48.2 Gbits/sec
[  1] 3.0000-4.0000 sec  5.64 GBytes  48.4 Gbits/sec
[  1] 4.0000-5.0000 sec  6.00 GBytes  51.6 Gbits/sec
[  1] 0.0000-5.0068 sec  28.5 GBytes  48.8 Gbits/sec
```

Figure - Throughput vs time when congestion algo is set as Cubic for h1, h2 and h3 respectively.

The same argument follows for the values that are seen in the above 3 figures. But one thing to notice is that the value in case of h1 went from 7.34Mbits/sec to 10.5Mbits/sec. Which does not seems like a linear increase nor looks like it doubled, it is non linear in nature. Thus, it can be said that TCP Cubic was used.

```
[  1] local 10.0.1.1 port 54966 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/5881)
tcpdump: listening on h1-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.25 MBytes  10.5 Mbits/sec
[  1] 1.0000-2.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 2.0000-3.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 3.0000-4.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 4.0000-5.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 0.0000-5.2105 sec  5.75 MBytes  9.26 Mbits/sec
```

```
[  1] local 10.0.1.2 port 38292 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/7615)
tcpdump: listening on h2-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.75 MBytes  14.7 Mbits/sec
[  1] 1.0000-2.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 2.0000-3.0000 sec  1.00 MBytes  8.39 Mbits/sec
[  1] 3.0000-4.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 4.0000-5.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 5.0000-5.7840 sec   128 KBytes  1.34 Mbits/sec
[  1] 0.0000-5.7840 sec  6.25 MBytes  9.06 Mbits/sec
```

```
tcpdump: listening on h3-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[  1] local 10.0.1.3 port 44888 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/4385)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  5.03 GBytes  43.2 Gbits/sec
[  1] 1.0000-2.0000 sec  5.63 GBytes  48.4 Gbits/sec
[  1] 2.0000-3.0000 sec  5.80 GBytes  49.8 Gbits/sec
[  1] 3.0000-4.0000 sec  5.89 GBytes  50.6 Gbits/sec
[  1] 4.0000-5.0000 sec  5.78 GBytes  49.7 Gbits/sec
[  1] 0.0000-5.0162 sec  28.1 GBytes  48.2 Gbits/sec
```

Figure - Throughput vs time when congestion algo is set as Vegas for h1, h2 and h3 respectively.

In case of congestion the values did not went down immediately to half rather the decline was smooth, indicating TCP vegas property.

```
[  1] local 10.0.1.1 port 53776 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/5739)
tcpdump: listening on h1-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.13 MBytes  9.44 Mbits/sec
[  1] 1.0000-2.0000 sec   768 KBytes  6.29 Mbits/sec
[  1] 2.0000-3.0000 sec  2.38 MBytes  19.9 Mbits/sec
[  1] 3.0000-4.0000 sec  1.07 MBytes  8.96 Mbits/sec
[  1] 4.0000-5.0000 sec   826 KBytes  6.76 Mbits/sec
[  1] 5.0000-6.9825 sec   126 KBytes   520 Kbits/sec
[  1] 0.0000-6.9825 sec  6.25 MBytes  7.51 Mbits/sec
```

```
[  1] local 10.0.1.2 port 45022 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/6593)
tcpdump: listening on h2-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  1.75 MBytes  14.7 Mbits/sec
[  1] 1.0000-2.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 2.0000-3.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 3.0000-4.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 4.0000-5.0000 sec  1.12 MBytes  9.44 Mbits/sec
[  1] 5.0000-6.0265 sec   128 KBytes  1.02 Mbits/sec
[  1] 0.0000-6.0265 sec  6.38 MBytes  8.87 Mbits/sec
```

```
[  1] local 10.0.1.3 port 55428 connected with 10.0.1.4 port 5001 (icwnd/mss
/irtt=14/1448/3834)
tcpdump: listening on h3-eth0, link-type EN10MB (Ethernet), snapshot length
262144 bytes
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  3.34 GBytes  28.7 Gbits/sec
[  1] 1.0000-2.0000 sec  4.17 GBytes  35.8 Gbits/sec
[  1] 2.0000-3.0000 sec  4.00 GBytes  34.4 Gbits/sec
[  1] 3.0000-4.0000 sec  4.17 GBytes  35.8 Gbits/sec
[  1] 4.0000-5.0000 sec  4.23 GBytes  36.3 Gbits/sec
[  1] 0.0000-5.0070 sec  19.9 GBytes  34.2 Gbits/sec
```

Figure - Throughput vs time when congestion algo is set as BBR for h1, h2 and h3 respectively.

D. Loss equal to 1% -

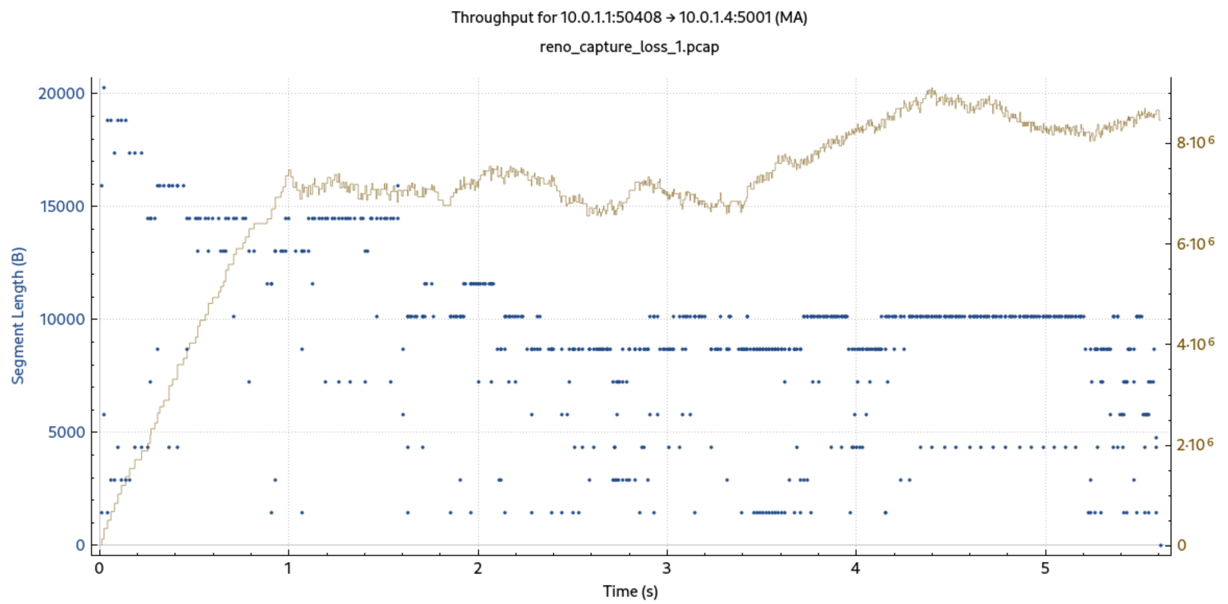


Figure - Throughput vs Time when congestion algorithm set to Reno and loss 1%

At first, the throughput increases monotonically but then it reaches its max window size and as a result the it starts to increase linearly. Until a packet loss is detected it continues to increase its window size.

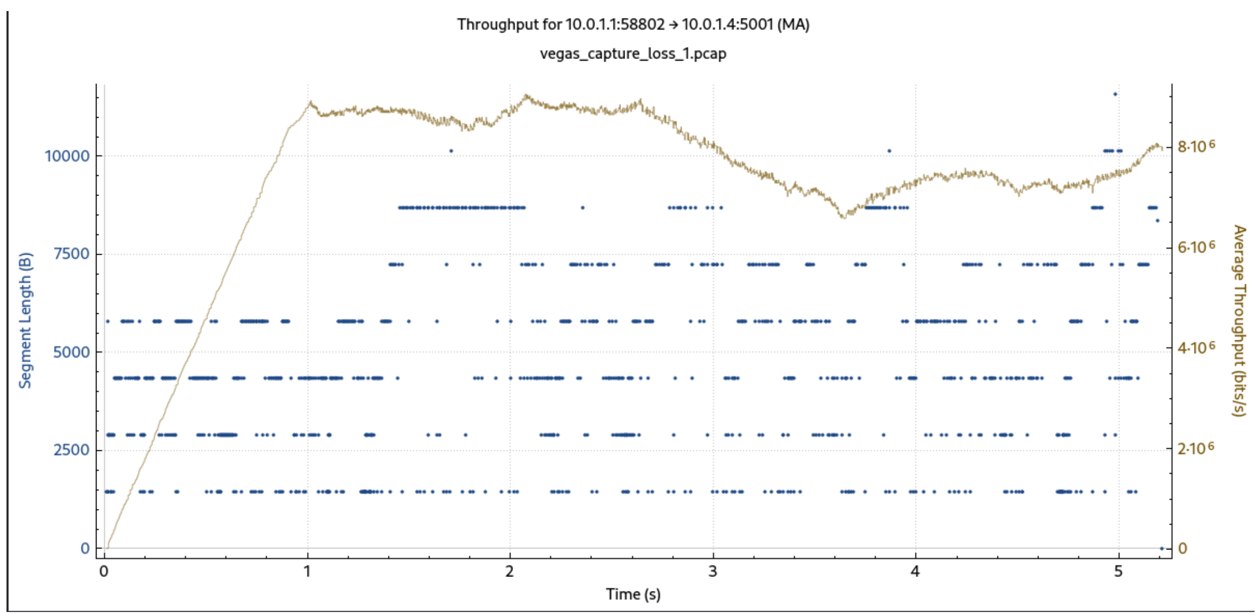


Figure - Throughput vs Time when congestion algorithm set to Vegas and loss 1%

In case of Vegas, the throughput is increased linearly at the beginning but when the congestion is touched it drops smoothly and then increases again. That is it dynamically increases/decreases its sending window size according to observed RTTs of sending packets.

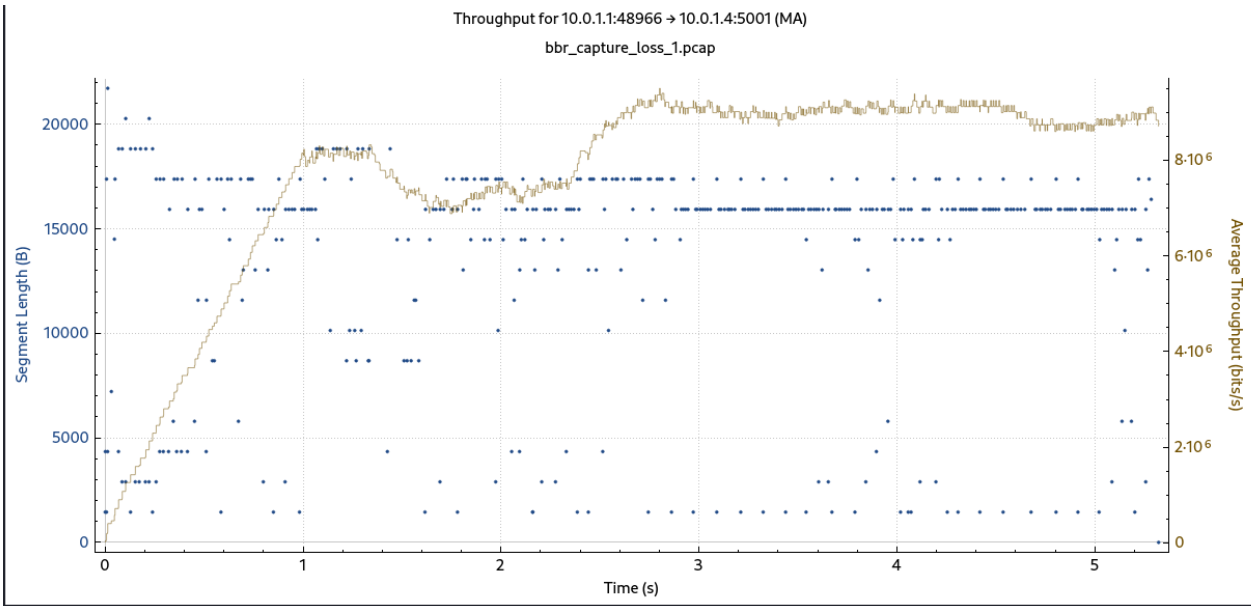


Figure - Throughput vs Time when congestion algorithm set to BBR and loss 1%

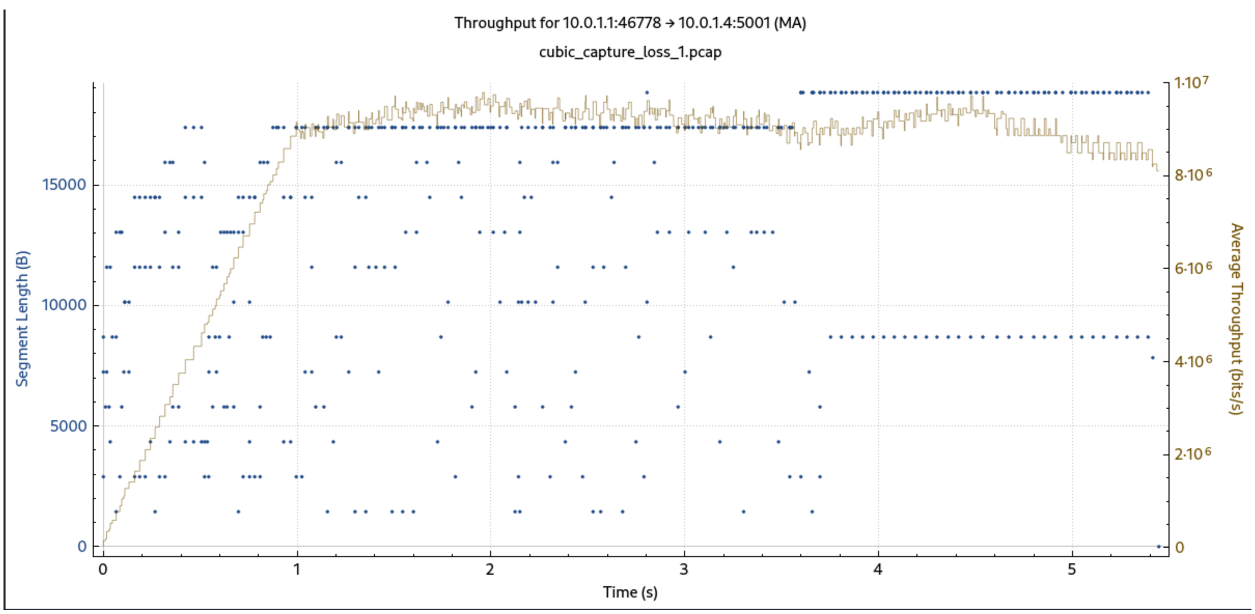


Figure - Throughput vs Time when congestion algorithm set to Cubic and loss 1%

In case of Cubic, the throughput is increased linearly at the beginning but then it reaches its threshold and starts to increase it in a polynomial manner(cubic) and hence the throughput continues to increase. Afterwards, at some point it decreases and then again increases rapidly resulting in less drop of throughput.

Loss equal to 3%

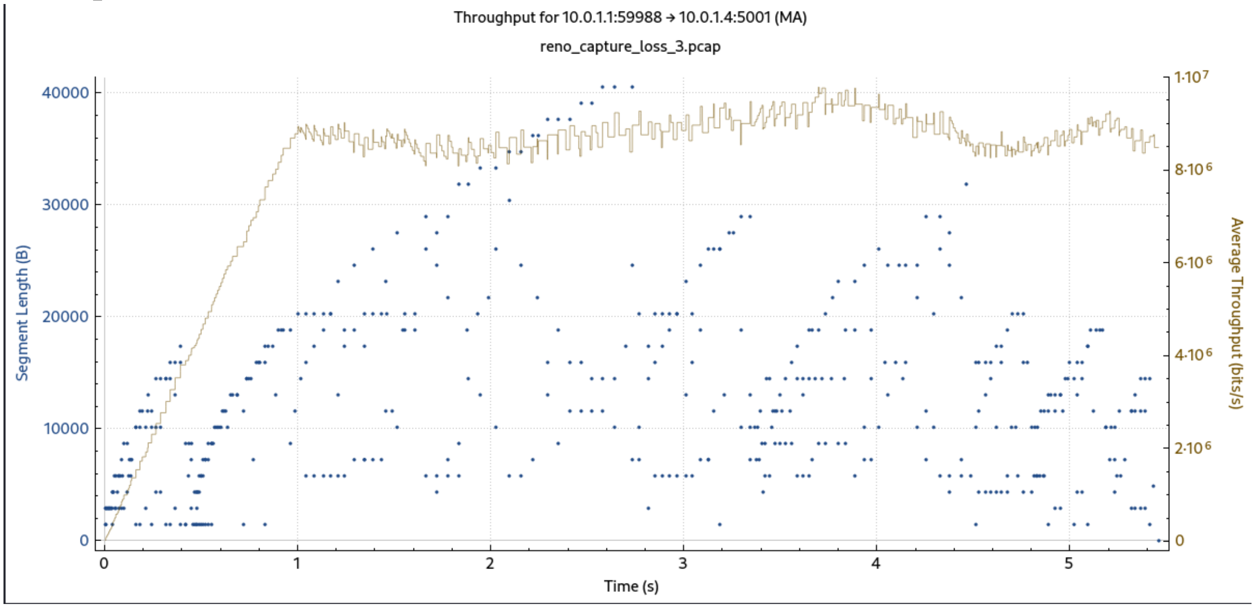


Figure - Throughput vs Time when congestion algorithm set to Reno and loss 3%

The fluctuation related to the MSS increased a lot when the loss is set to 3%. It increased and decreased, but whenever it decreased it the throughput is decreased only a little as it instaly increases the throughput by doubling the number of packets that are sent to the server.

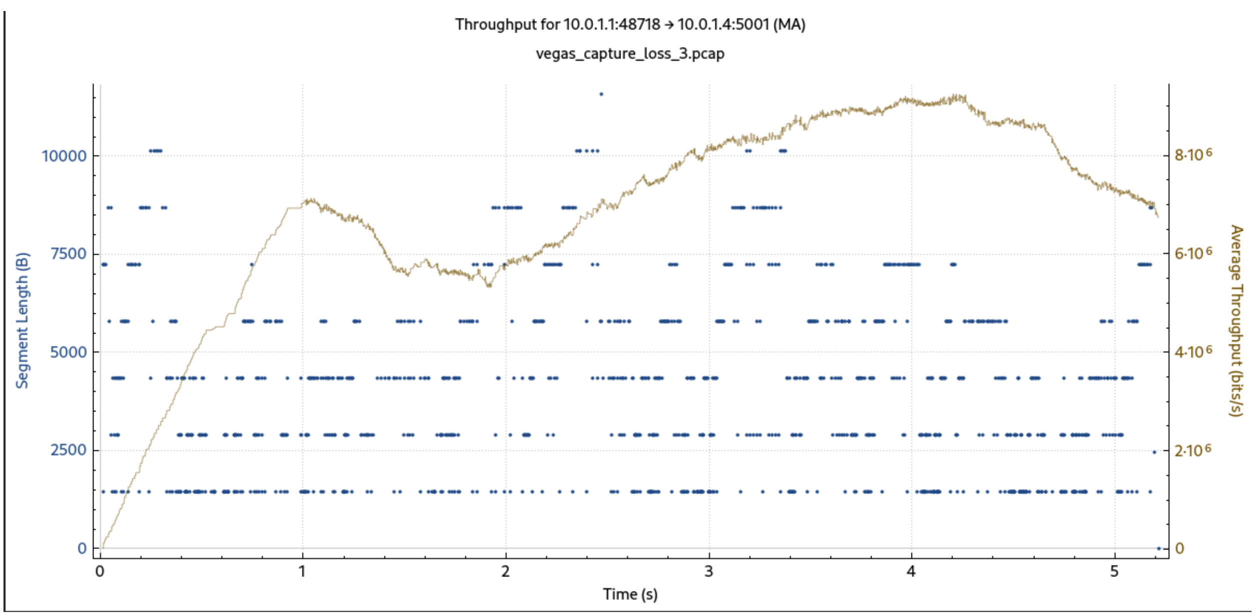


Figure - Throughput vs Time when congestion algorithm set to Vegas and loss 3%

At 3% loss, the shape of the curve resembles more to that of the actual graph, i.e., it first increases and based on the value of RTT it either keeps on increasing or decreasing the number of packets sent. As soon as the RTT increases the packet sending rate decreases.

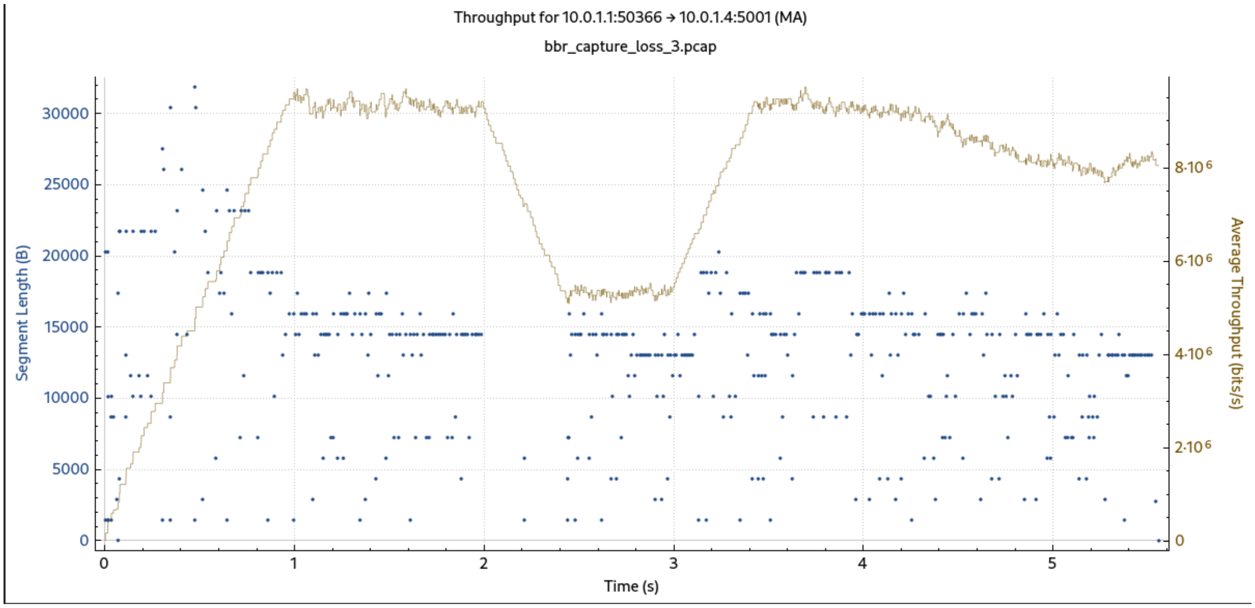


Figure - Throughput vs Time when congestion algorithm set to BBR and loss 3%

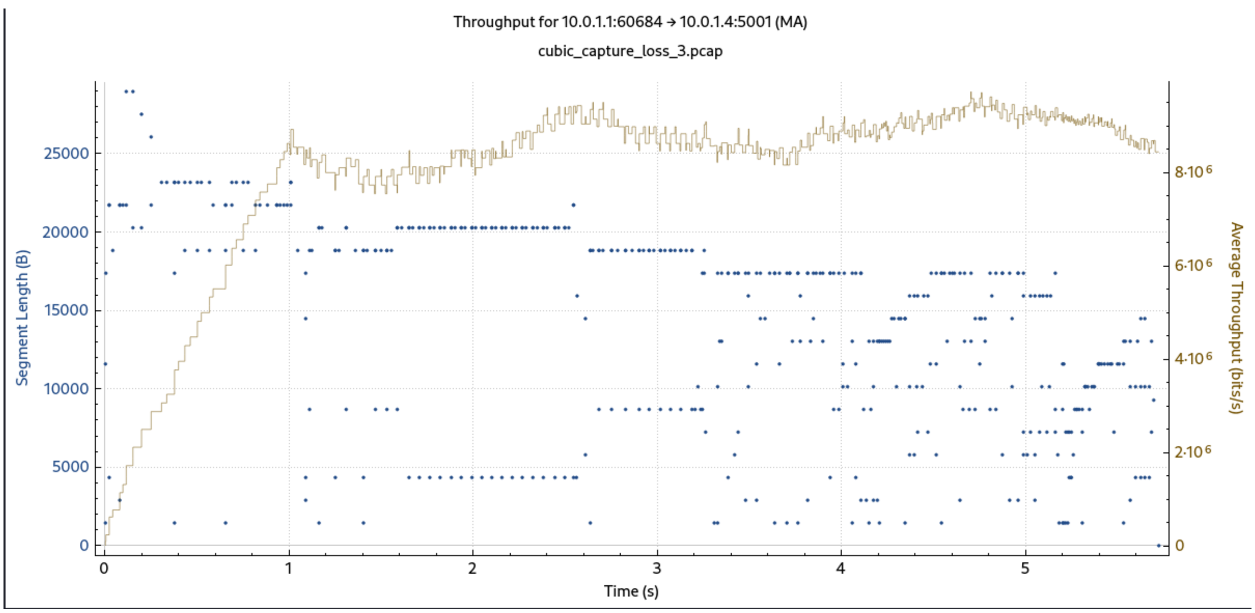


Figure - Throughput vs Time when congestion algorithm set to Cubic and loss 3%

In case of Cubic, it reached the threshold and then increased faster then TCP Reno by sending more packets at the beginning of crossing threshold and then limiting the number of packets as it more closer and closer to the congestion portion.

References:

1. https://github.com/mininet/mininet/blob/master/examples/linuxrouter.py
2. https://stackoverflow.com/questions/46595423/mininet-how-to-create-a-topology-with-two-routers-and-their-respective-hosts#:~:text=company%20blog-,(mininet)%20How%20to%20create%20a%20topology%20with%20two,routers%20and%20their%20respective%20hosts&text=The%20scenario%20is%20composed%20by,subnet%2C%20the%20net%20ping%20properly
3. http://mininet.org/walkthrough/
4. https://iperf.fr/iperf-doc.php