FINAL SUBMISSION

Assignment-02 **CS 433 Computer Networks (2023-24)** **10-11-2023**

Group Members:
1) Mithil Pechimuthu - 21110129
2) Kaushal Kothiya - 21110107

GitHub repository ➜https://github.com/PechimuthuMithil/CS433_Assignment-2

Part I: Implement the routing functionality in mininet.

Basic Instructions:

Make sure to start Openv Switch ➜ sudo systemctl start openvswitch-switch.service

Run `Part1.py` ➜ sudo python3 Part1.py

Implementation :

Please find the program here.

The network topology that we have implemented and emulated is shown in Figure 1.

ra-eth2 = 10.100.1.10

ra-eth3 = 10.101.2.10

rb-eth2 = 10.100.1.20

rb-eth3 = 10.102.3.10

rc-eth2 = 10.101.2.20

rc-eth3 = 10.102.3.20

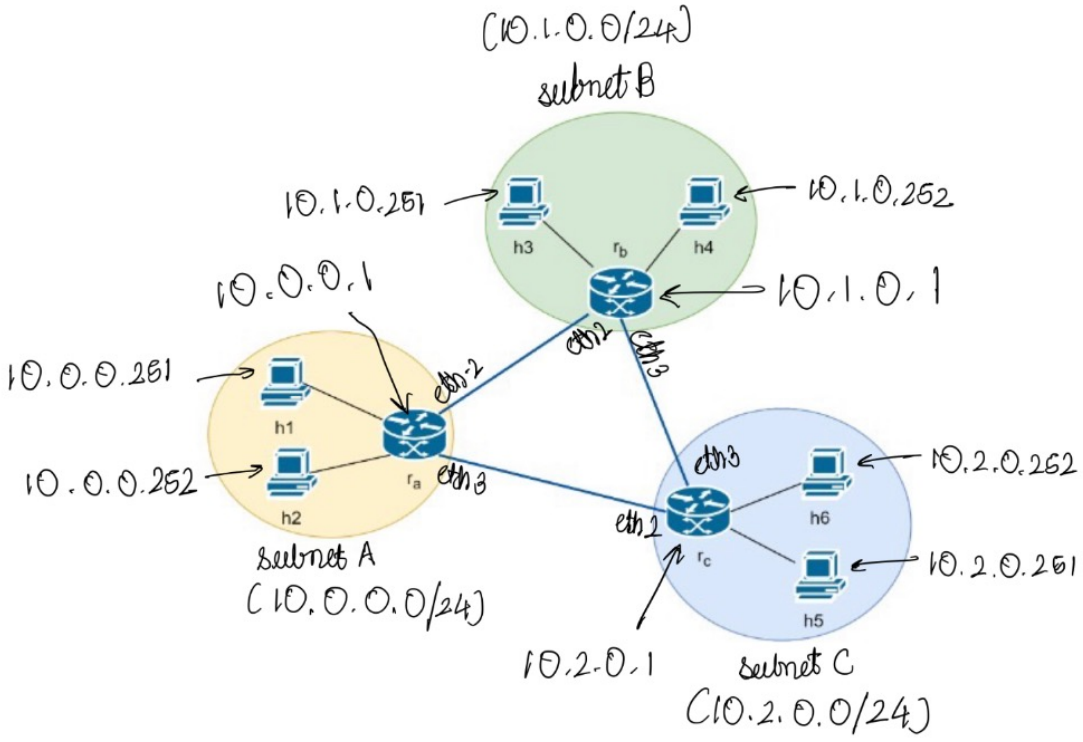


Figure 1: The implemented network topology

The python program is just to emulate the topology. To see the results as the question demands, it is required that one follow the instructions and type the corresponding commands. Xterms are manually opened to see routing tables and manipulate them. Screenshots of outputs and steps have also been provided to help the reader.

To show that every host is able to send packets to every other host, we tested the topology using the '*pingall*' command (Figure 2).

```
┌──(kali㉿kali)-[~/Desktop/Assignment2]
└─$ sudo python3 Part1.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) (ra, rc) (rb, rc) (s1, ra) (s2, rb) (s3, rc)
*** Configuring hosts
h1 h2 h3 h4 h5 h6 ra rb rc
*** 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> links
```

Figure 2: Output of pingall.

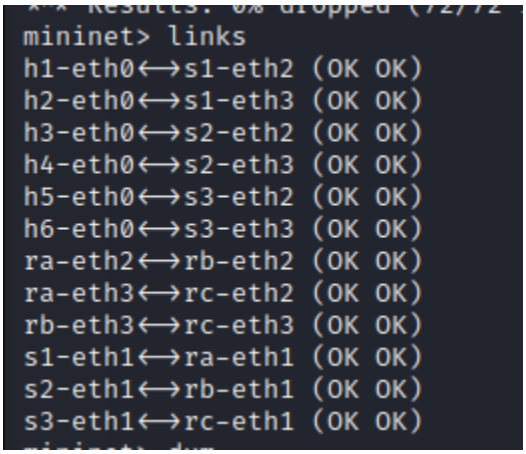


Figure 3: The 9 links in the topology shown in figure + 3 links between each subnet switch to the corresponding router.

Observations:

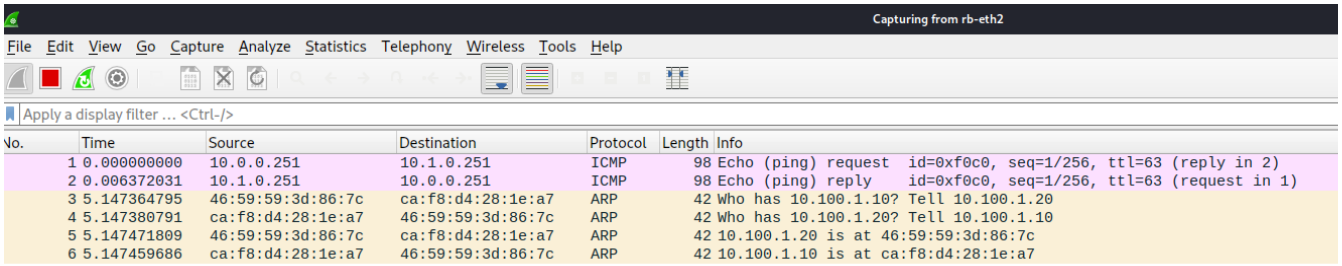


Figure 4: Packets through router (rb-eth2) while pinging h1 to h3.

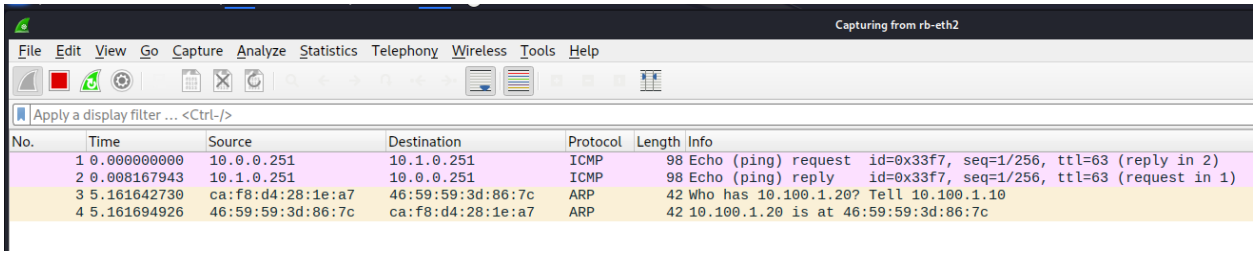


Figure 5: Packets through router (rb-eth2) while pinging h1 to h4.

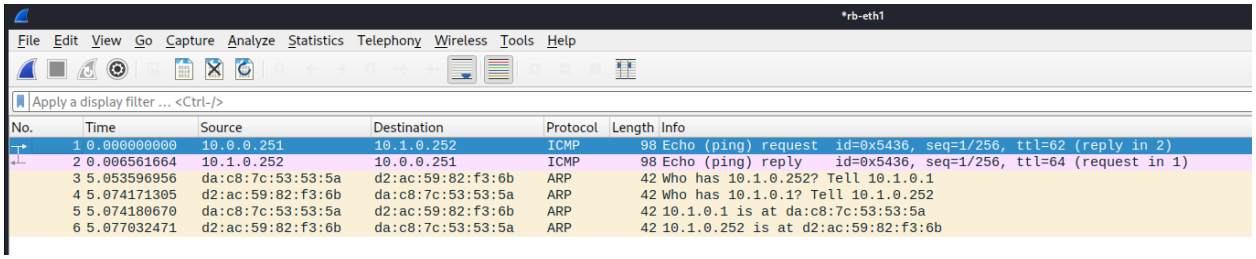


Figure 6: Packet flow through router (rb-eth1)

Changing routing path:

a. h1 -> r_a -> r_c -> h6:

This is the default path. Figure 7 shows this routing that we can obtain by running traceroute 10.2.0.252 from h1.

```
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# traceroute 10.2.0.252
traceroute to 10.2.0.252 (10.2.0.252), 30 hops max, 60 byte packets
 1  10.0.0.1 (10.0.0.1)  7.239 ms  6.886 ms  18.010 ms
 2  10.101.2.20 (10.101.2.20)  17.454 ms  17.071 ms  16.693 ms
 3  10.2.0.252 (10.2.0.252)  22.813 ms  22.431 ms  22.030 ms

┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─#
```

Figure 7: traceroute 10.2.0.252. ra-eth1(10.0.0.1), rc-eth2(10.101.2.20), h6(10.2.0.252)

Figure 8 shows us that average RTT in this routing is *0.131 ms*

```
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# ping 10.2.0.252 -c10
PING 10.2.0.252 (10.2.0.252) 56(84) bytes of data.
64 bytes from 10.2.0.252: icmp_seq=1 ttl=62 time=0.347 ms
64 bytes from 10.2.0.252: icmp_seq=2 ttl=62 time=0.070 ms
64 bytes from 10.2.0.252: icmp_seq=3 ttl=62 time=0.055 ms
64 bytes from 10.2.0.252: icmp_seq=4 ttl=62 time=0.068 ms
64 bytes from 10.2.0.252: icmp_seq=5 ttl=62 time=0.072 ms
64 bytes from 10.2.0.252: icmp_seq=6 ttl=62 time=0.148 ms
64 bytes from 10.2.0.252: icmp_seq=7 ttl=62 time=0.114 ms
64 bytes from 10.2.0.252: icmp_seq=8 ttl=62 time=0.075 ms
64 bytes from 10.2.0.252: icmp_seq=9 ttl=62 time=0.224 ms
64 bytes from 10.2.0.252: icmp_seq=10 ttl=62 time=0.137 ms

--- 10.2.0.252 ping statistics ---
10 packets transmitted, 10 received, 0% packet loss, time 9229ms
rtt min/avg/max/mdev = 0.055/0.131/0.347/0.087 ms
```

Figure 8: Calculating latency using average RTT.

```
"Node: h1"
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# iperf -c 10.2.0.252 -p 5001 -t 10
------------------------------------------------------------
Client connecting to 10.2.0.252, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.251 port 52500 connected with 10.2.0.252 port 5001 (icwnd/mss
/irtt=14/1448/4971)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-10.0190 sec  39.0 GBytes  33.5 Gbits/sec
```

By running h6 as a server and h1 as a client, we can use iperf to see the bandwidth of the connection. For the default route, it is 33.5 Gbits/sec. See figure given above.

b. h1-> r_a -> r_b -> r_c -> h6

We can run the following command in ra:

'*ip route change 10.2.0.0/24 via 10.100.1.10 dev ra-eth2*' to make the packets destined to subnet 10.2.0.0/24 via rb.

Also run the following command in rc.

'i*p route change 10.0.0.0/24 via 10.102.3.10 dev rc-eth3*' to make the packets destined to 10.0.0.0/24 subnet from rc subnet go through rb.

Figure 9 shows this routing that we can obtain by running traceroute 10.2.0.252 from h1.

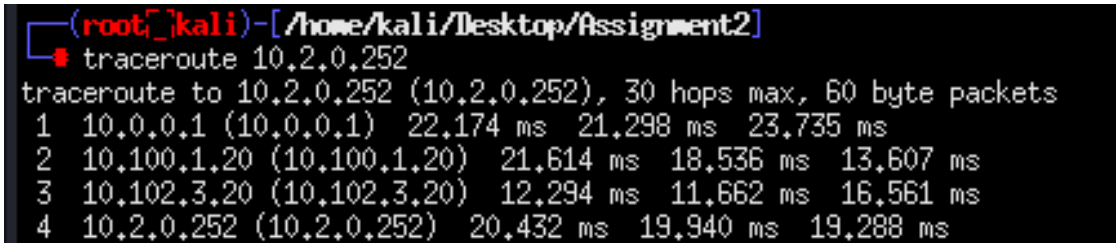


Figure 9: traceroute 10.2.0.252. ra-eth1(10.0.0.1), rb-eth2(10.101.2.20), rc-eth3(10.102.3.20) h6(10.2.0.252)

Figure 10 shows us that average RTT in this routing is *0.215 ms*

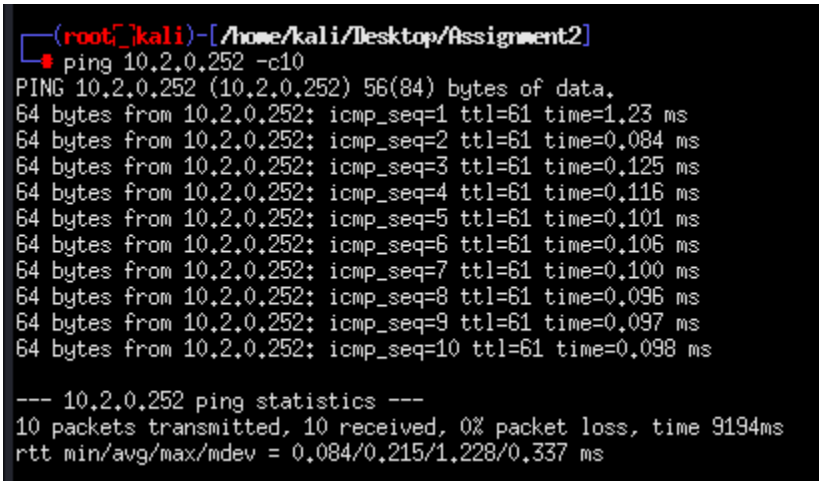
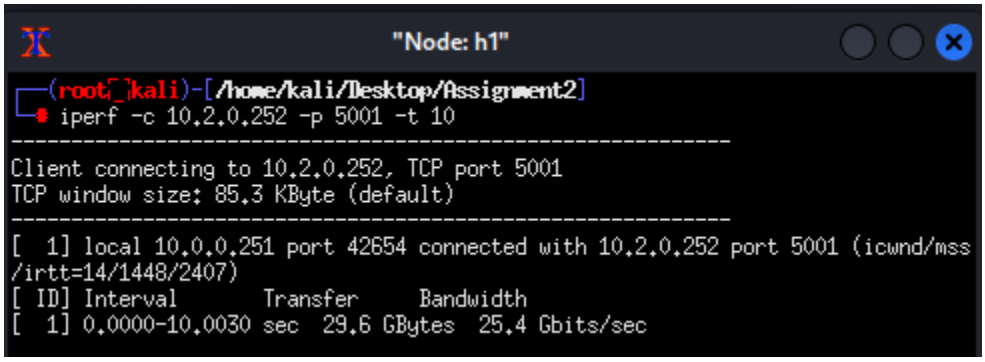


Figure 10: Calculating latency using average RTT for new routing.



By running h6 as a server and h1 as a client, we can use iperf to see the bandwidth of the connection. For the new route, it is 25.4 Gbits/sec. See figure given above.

This value changes often depending on the various parameters that change during execution.

Routing tables:

Part a:

Router ra

Router rb

Router rc

```
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# ip route
10.0.0.0/24 via 10.101.2.10 dev rc-eth2
10.1.0.0/24 via 10.102.3.10 dev rc-eth3
10.2.0.0/24 dev rc-eth1 proto kernel scope link src 10.2.0.1
10.101.2.0/24 dev rc-eth2 proto kernel scope link src 10.101.2.20
10.102.3.0/24 dev rc-eth3 proto kernel scope link src 10.102.3.20
```

Part c

Router ra

```
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# ip route
10.0.0.0/24 dev ra-eth1 proto kernel scope link src 10.0.0.1
10.1.0.0/24 via 10.100.1.20 dev ra-eth2
10.2.0.0/24 via 10.100.1.20 dev ra-eth2
10.100.1.0/24 dev ra-eth2 proto kernel scope link src 10.100.1.10
10.101.2.0/24 dev ra-eth3 proto kernel scope link src 10.101.2.10
```

Router rb

```
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# ip route
10.0.0.0/24 via 10.100.1.10 dev rb-eth2
10.1.0.0/24 dev rb-eth1 proto kernel scope link src 10.1.0.1
10.2.0.0/24 via 10.102.3.20 dev rb-eth3
10.100.1.0/24 dev rb-eth2 proto kernel scope link src 10.100.1.20
10.102.3.0/24 dev rb-eth3 proto kernel scope link src 10.102.3.10
```

Router rc

```
┌──(root㉿kali)-[/home/kali/Desktop/Assignment2]
└─# ip route
10.0.0.0/24 via 10.102.3.10 dev rc-eth3
10.1.0.0/24 via 10.102.3.10 dev rc-eth3
10.2.0.0/24 dev rc-eth1 proto kernel scope link src 10.2.0.1
10.101.2.0/24 dev rc-eth2 proto kernel scope link src 10.101.2.20
10.102.3.0/24 dev rc-eth3 proto kernel scope link src 10.102.3.20
```

References:

Following links were referred to understand the working of routers and how to connect two of them.

1) https://stackoverflow.com/questions/46595423/mininet-how-to-create-a-topolog

y-with-two-routers-and-their-respective-hosts

2) https://stackoverflow.com/questions/70160081/how-do-i-connect-three-routers-to-three-hosts-in-mininet
3) https://github.com/mininet/mininet/blob/master/examples/linuxrouter.py

---

Part II: Throughput for different congestion control schemes

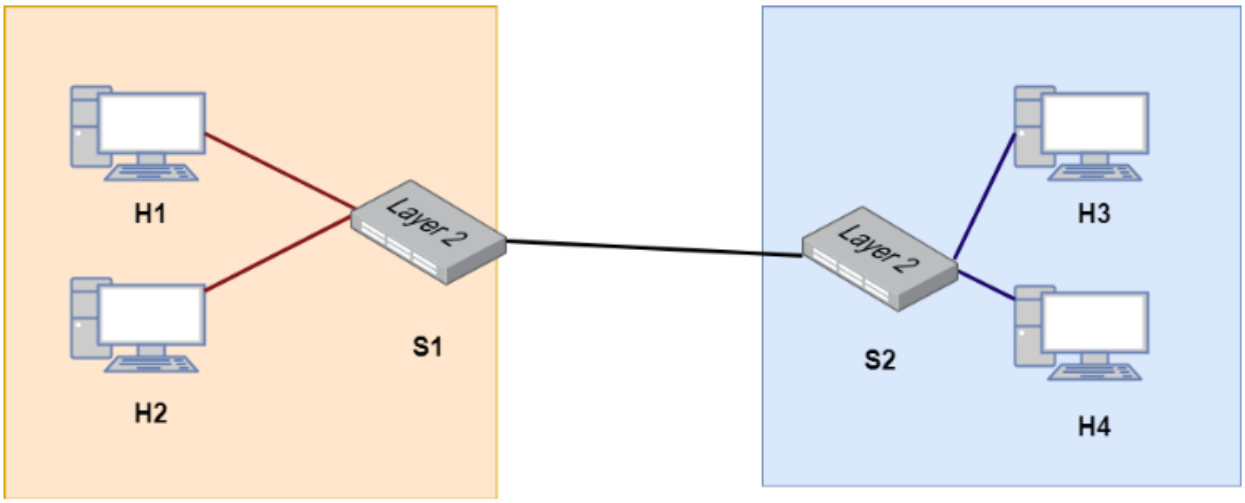


Figure 11: The emulated network topology

Basic Instructions:

Make sure to start Openv Switch ➔ sudo systemctl start openvswitch-switch.service

Run `Part1.py` ➔ sudo python3 Part2.py --congestion=reno --config=c --linkloss=2

Implementation:

Please find the program for Part2 here.

The program emulates the topology shown in Figure 11. The program generates packet capture files upon running that are later used for analysis using wireshark. The program will take the arguments you provide to it and then modify the network parameters such as congestion mode, link loss accordingly and emulate what is desired. To generate TCP packets, the program uses iperf [2].

Throughput analysis as mentioned previously is done through wireshrak with the help of

the packet capture files.

We used TCLinks to make a realistic network, and used tcpdump for 10-15s to capture the packets transmitted between clients and server. These captured packets were put in a pcap file that is later analyzed using wireshark.

Observations:

We know that segment length = min(congestion window, receive window).
Assuming the receive window is large enough, we can say that initially, or mostly the segment length corresponds to the congestion window. Sometimes when the receive window (which was by default set as 85.3 kB) becomes small, then we see constant segment size.

Client on H1 and the server on H4

Report:

a. reno

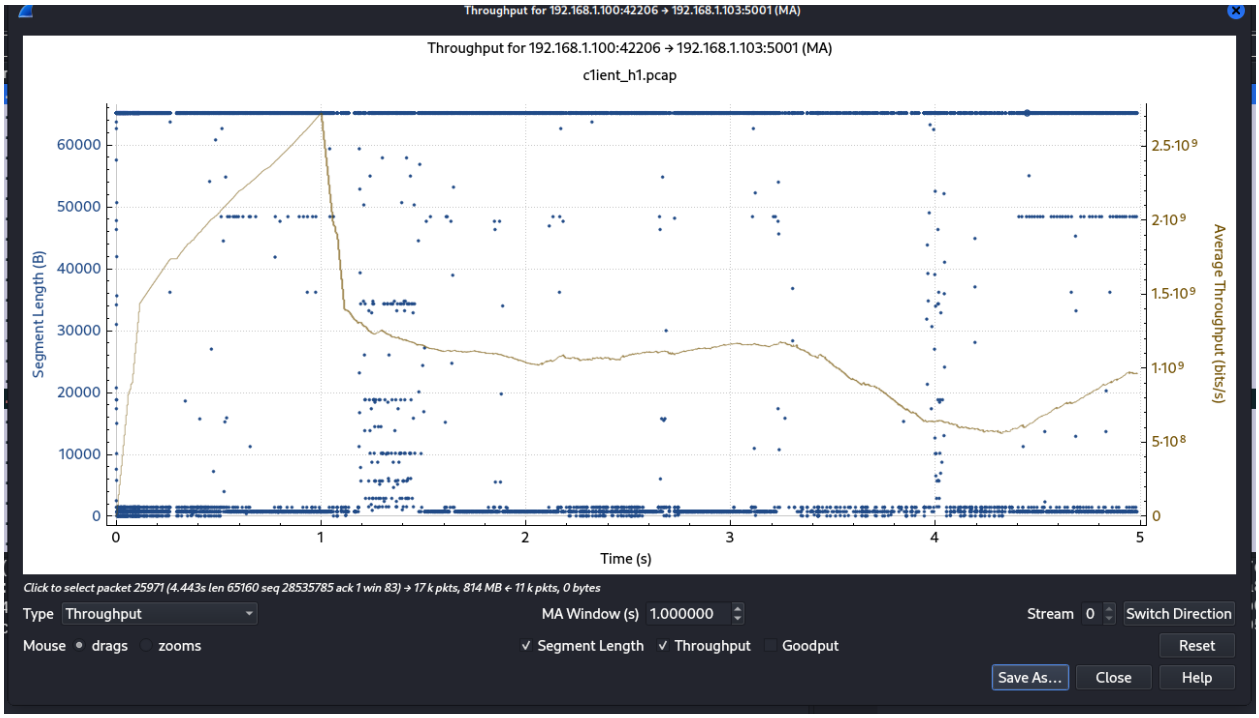


Figure 12: Throughput of the h1

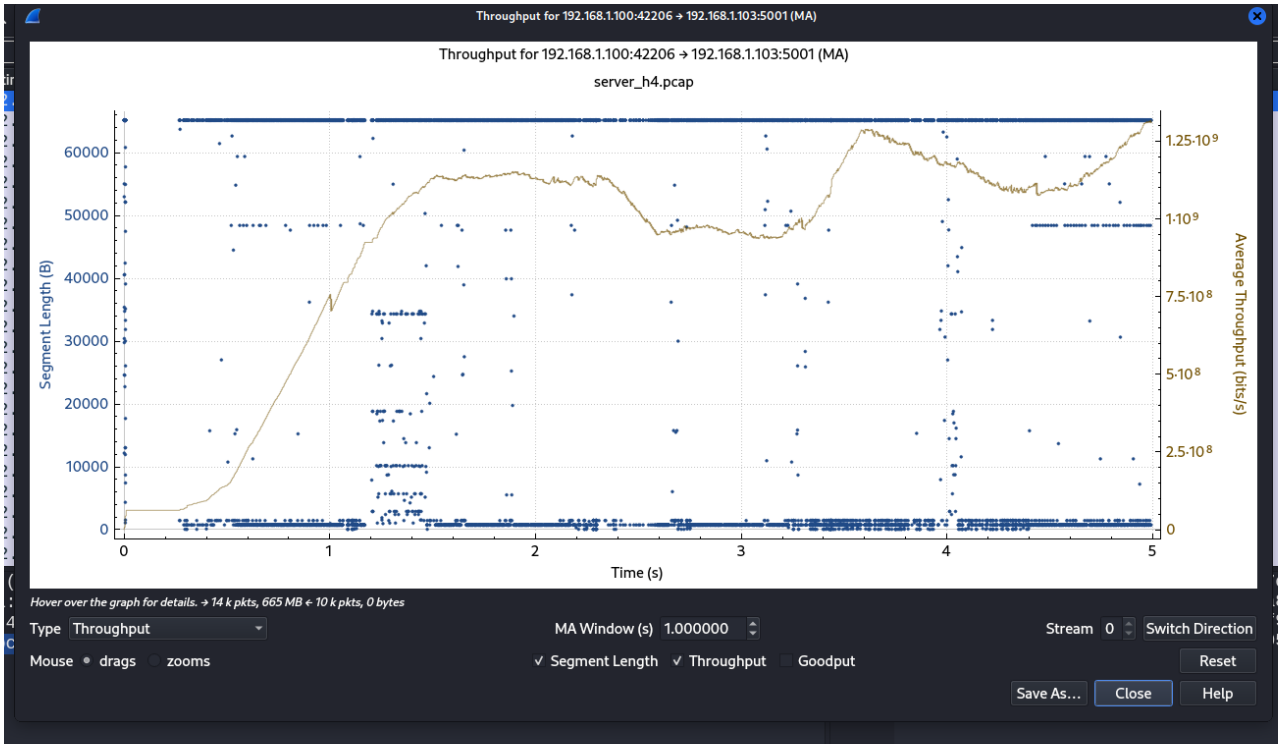


Figure 13: Throughput of server h4

Reason: The TCP Reno Saw tooth graph is seen in Fig 12 because of the slow start and congestion avoidance phase. After that the effect of a small receive buffer takes over

b. Vegas

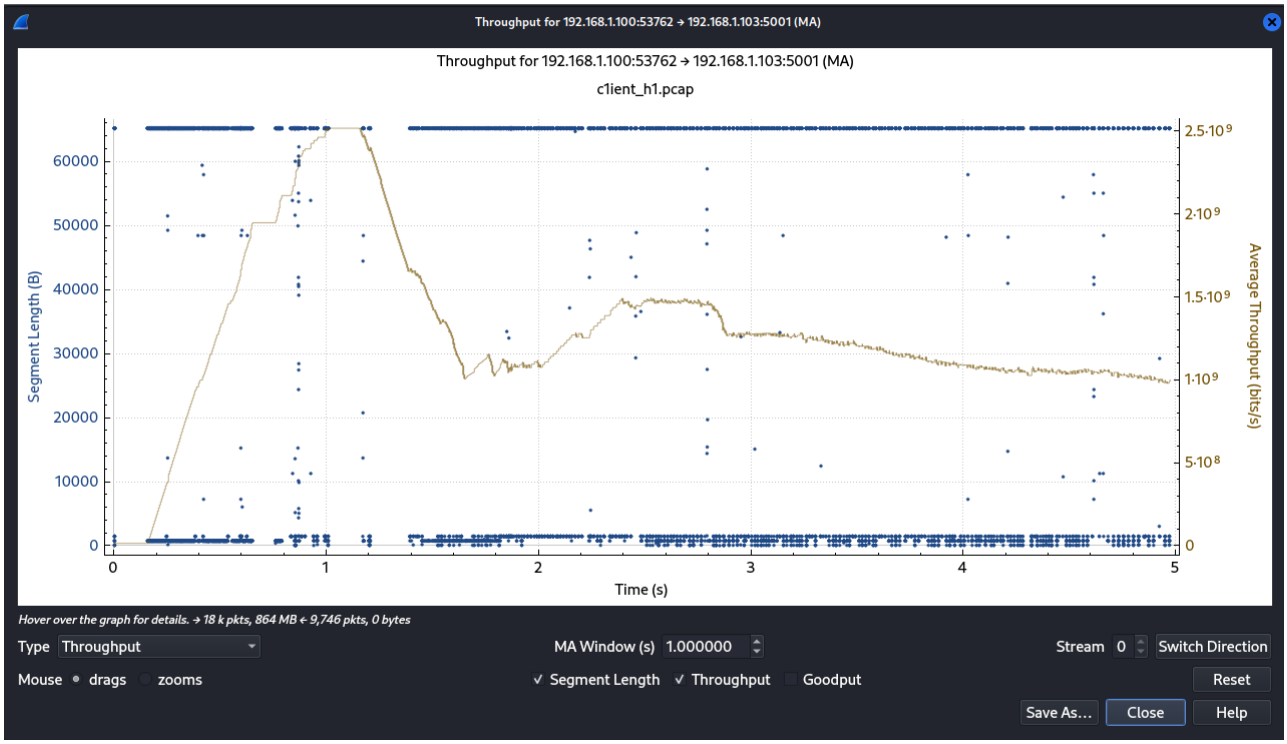


Figure 14: Throughput of the h1

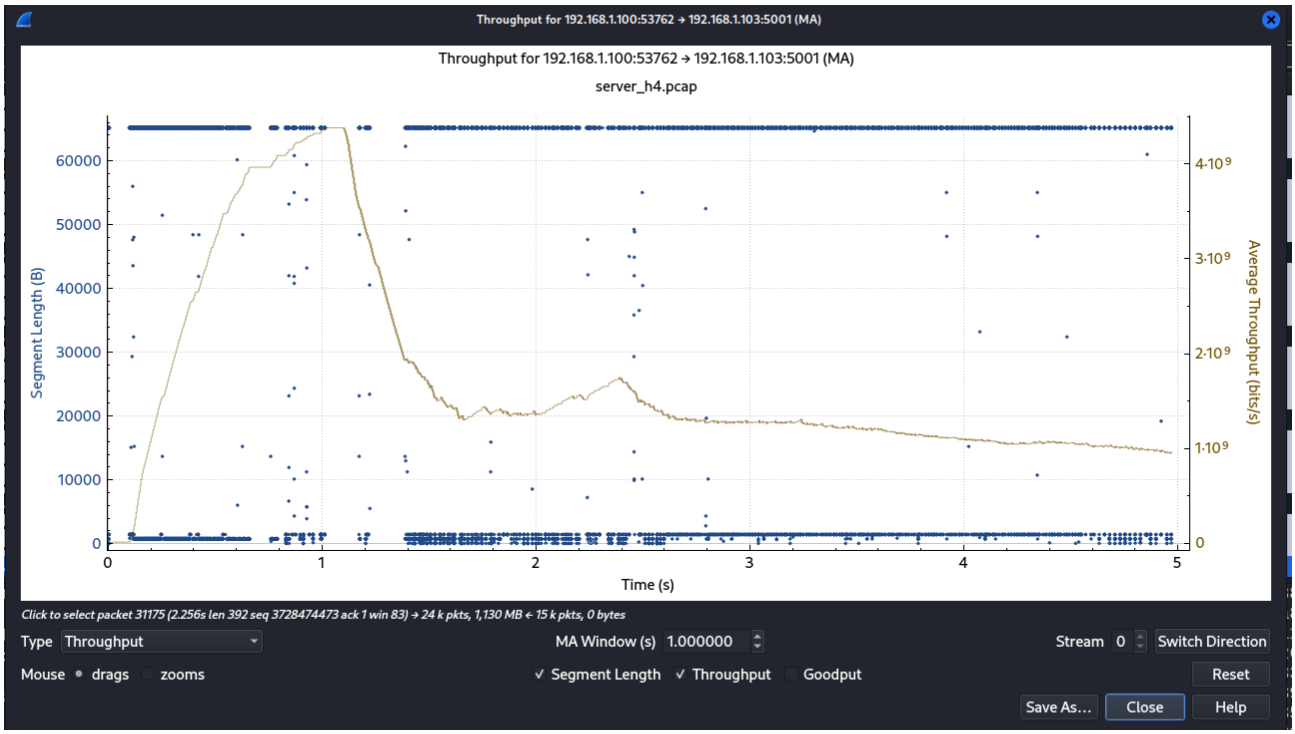


Figure 15: Throughput of server h4

Reason: TCP vegas detects congestion based on the difference between actual and

expected RTT. This is able to detect congestion before it occurs, which is different from Reno, and thus we can see the corresponding higher throughput. Later after peeking at 1 second, there seems to be congestion that drastically reduces the throughput. Later we see stagnation of the throughput because of a filled receive buffer.

c. Cubic

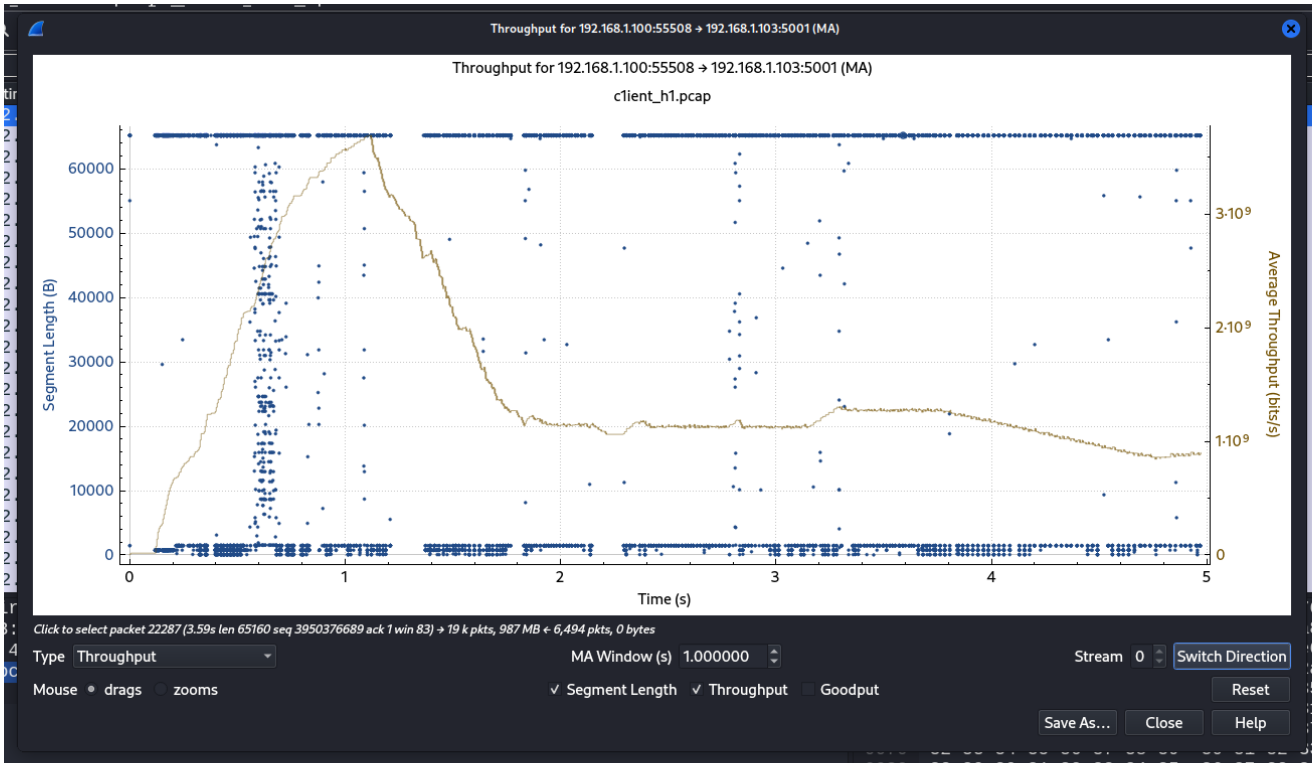


Figure 16: Throughput of the h1

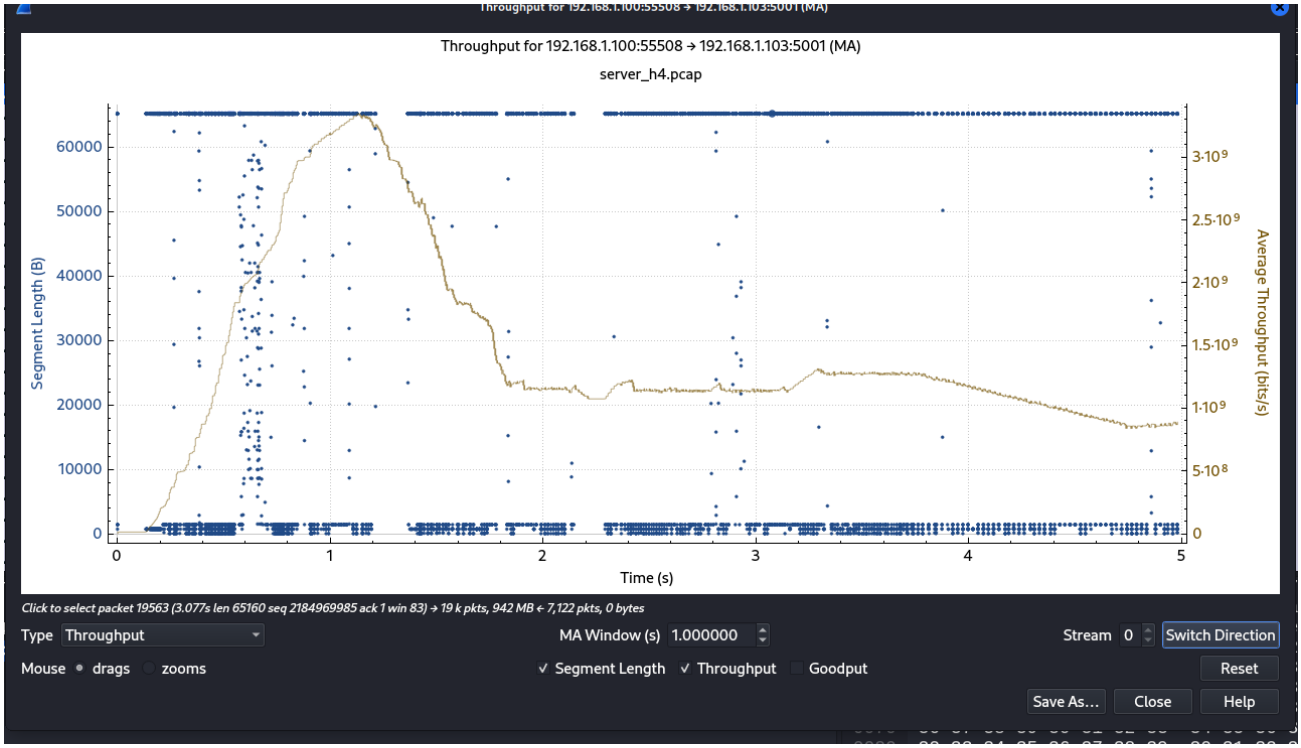


Figure 17: Throughput of server h4

Reason: In TCP Cubic we are able to see that we can reach the peak throughput before loss much faster than in TCP Reno. We can see the improvements due to the cubic increase in the slow start phase.

d. BBR

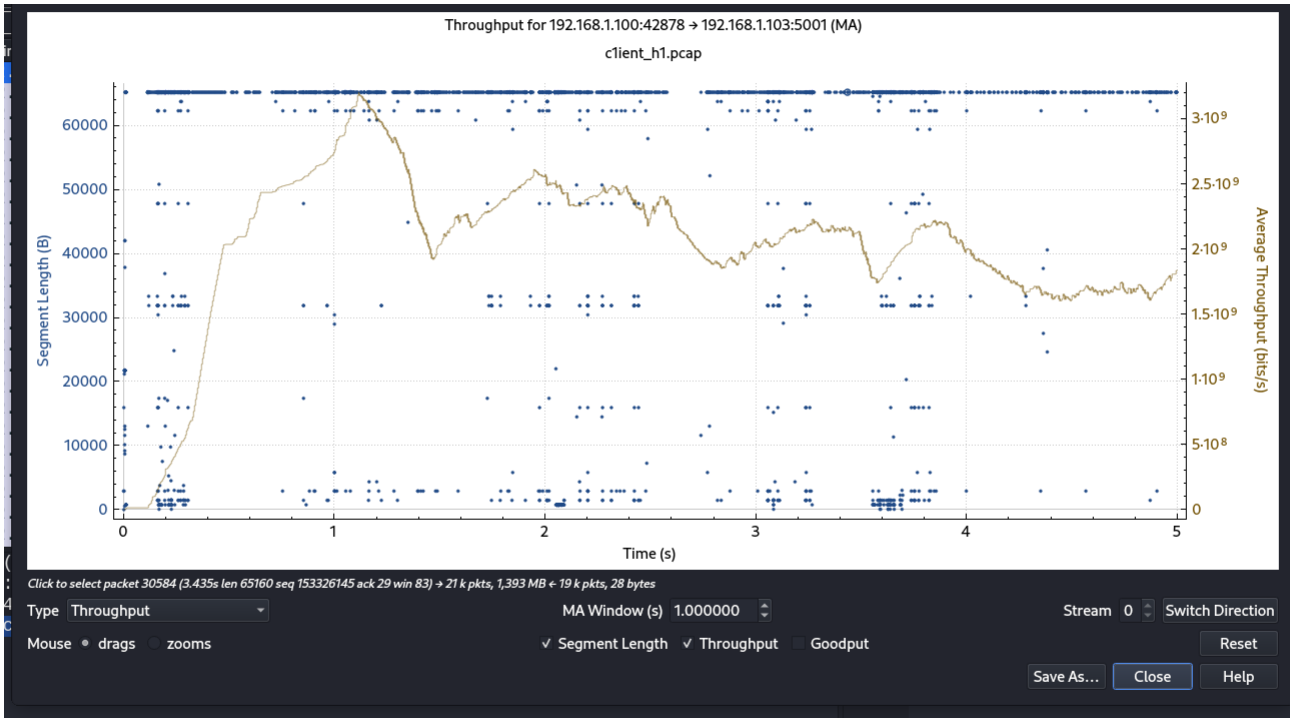


Figure 18: Throughput of the h1

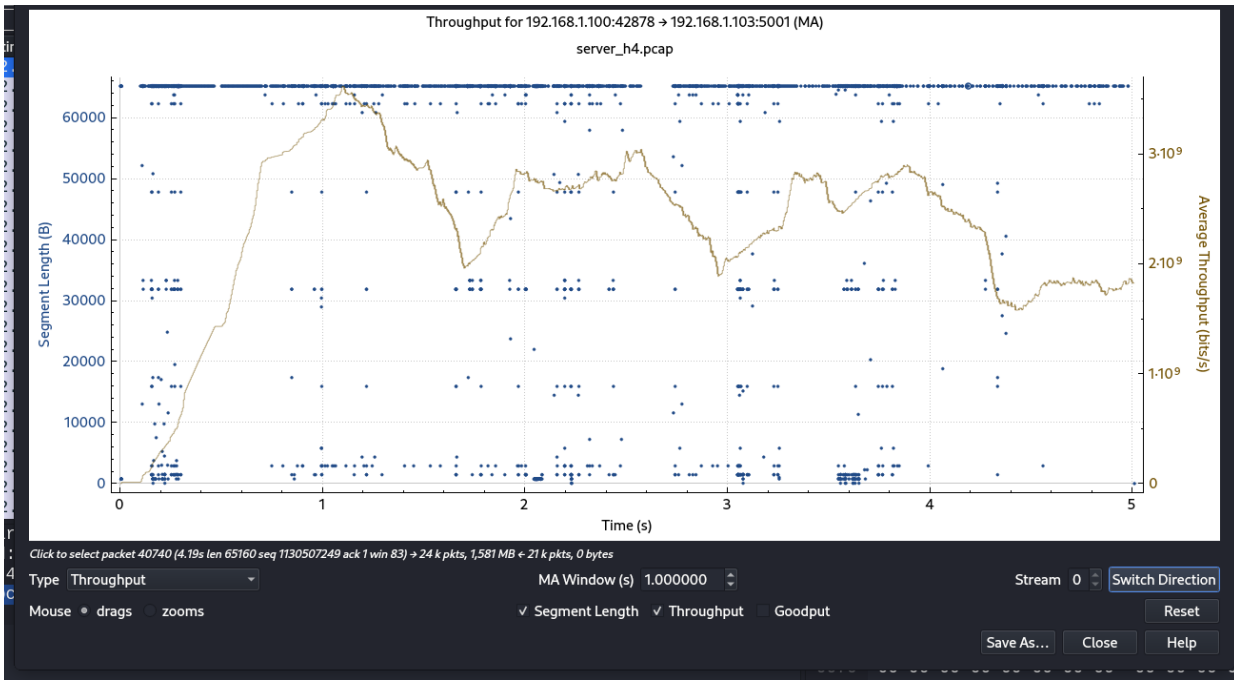


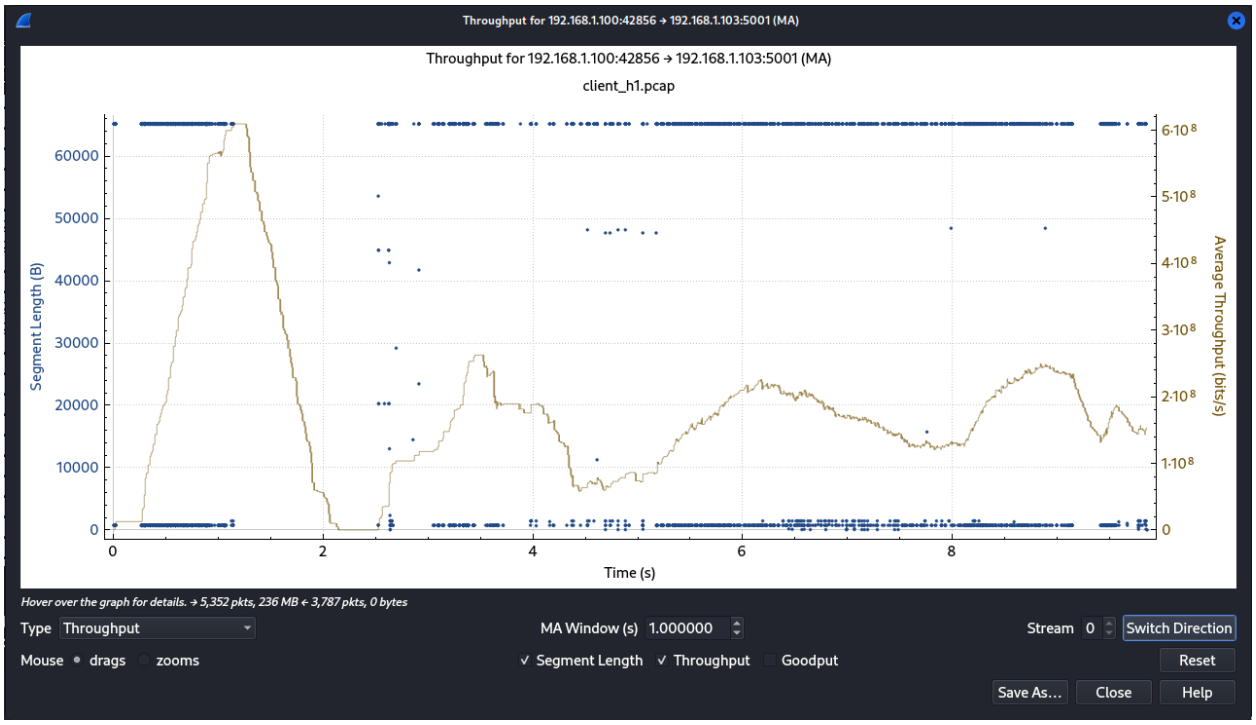
Figure 19: Throughput of server h4

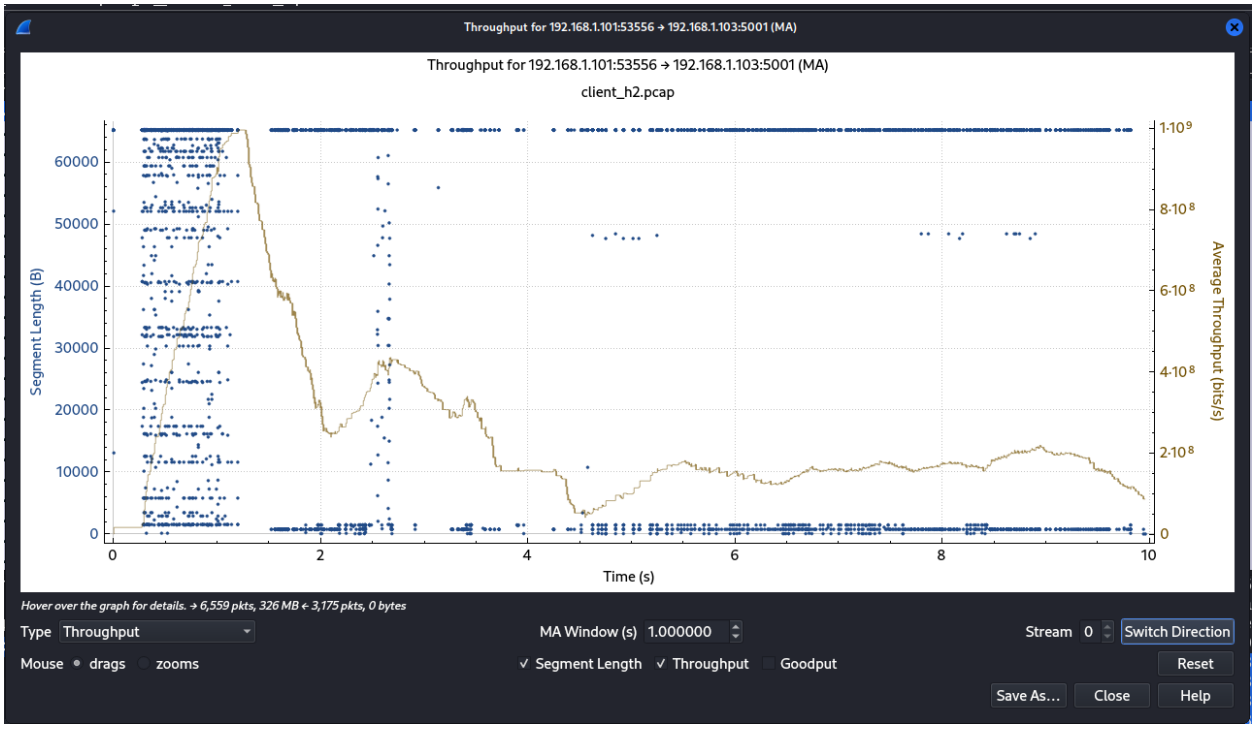
Reason: TCP BBR is also another rate/ delay based congestion control like TCP Vegas. However, this is more smart in deciding the speed of sending data and data in the

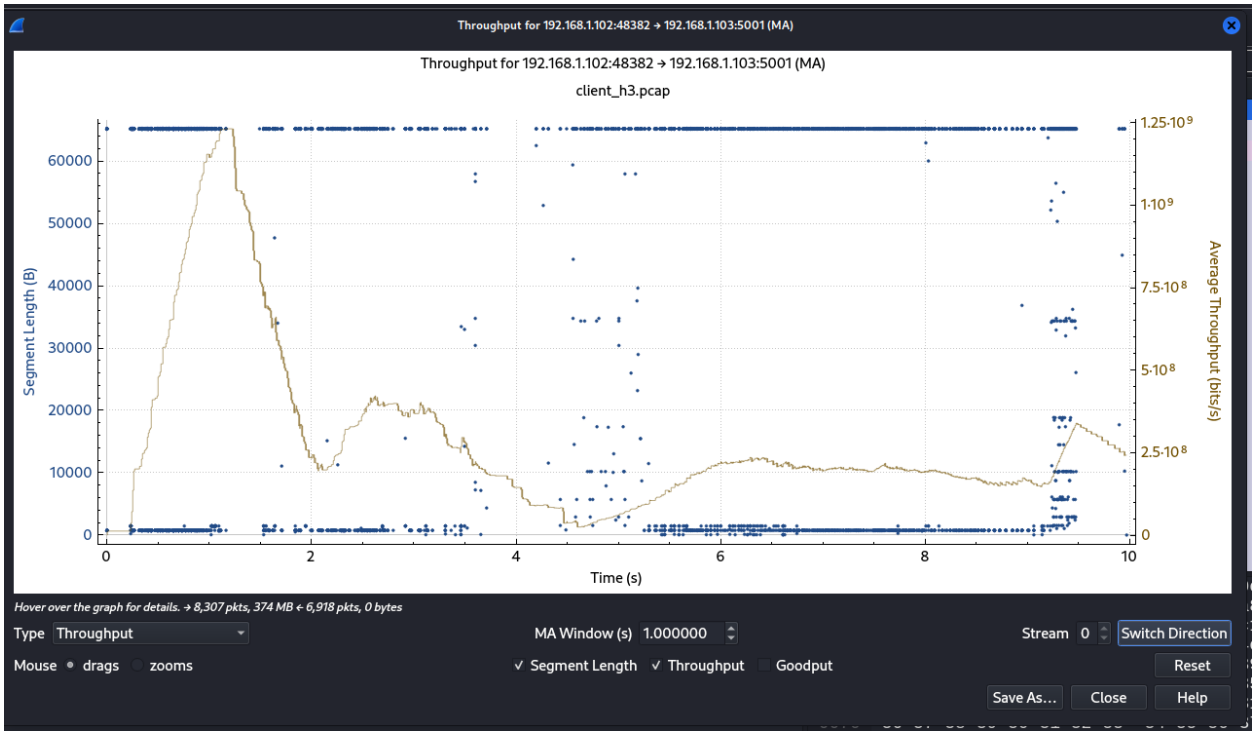
network. Hence we see the consistent high throughput as compared to others that peaked and then fell to constant minimum.

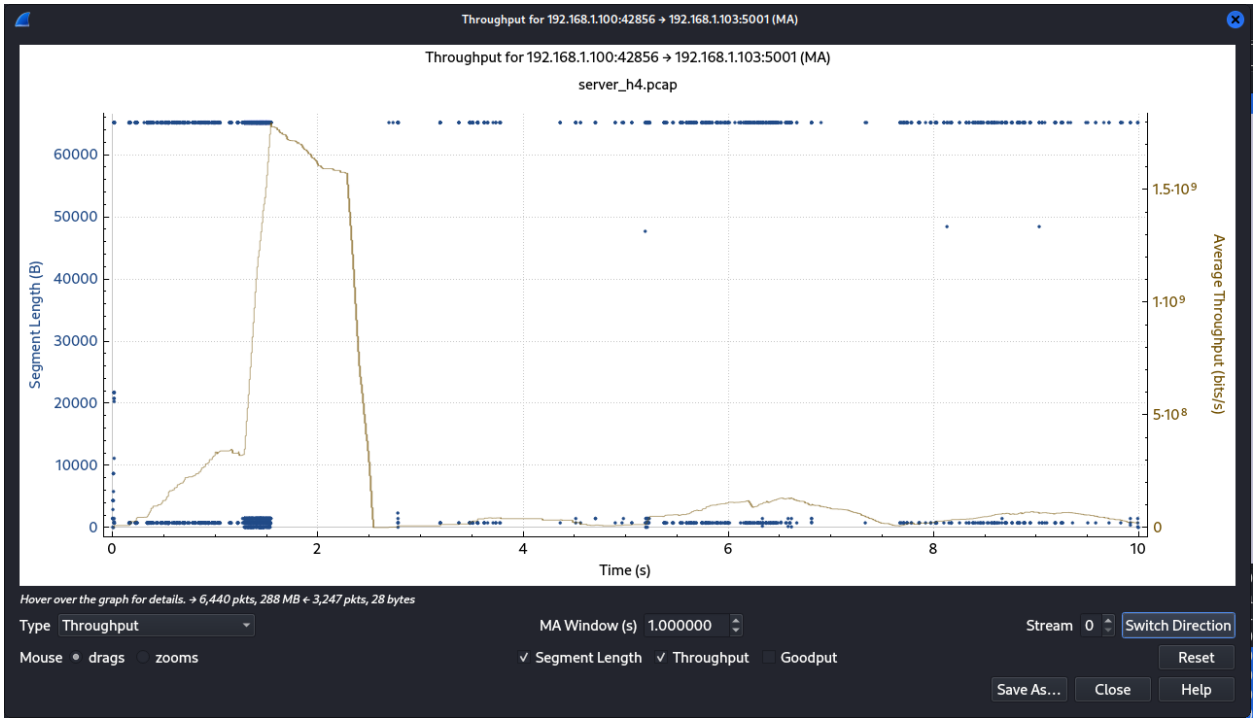
Client on H1, H2, H3 simultaneously and the server on H4

a. reno

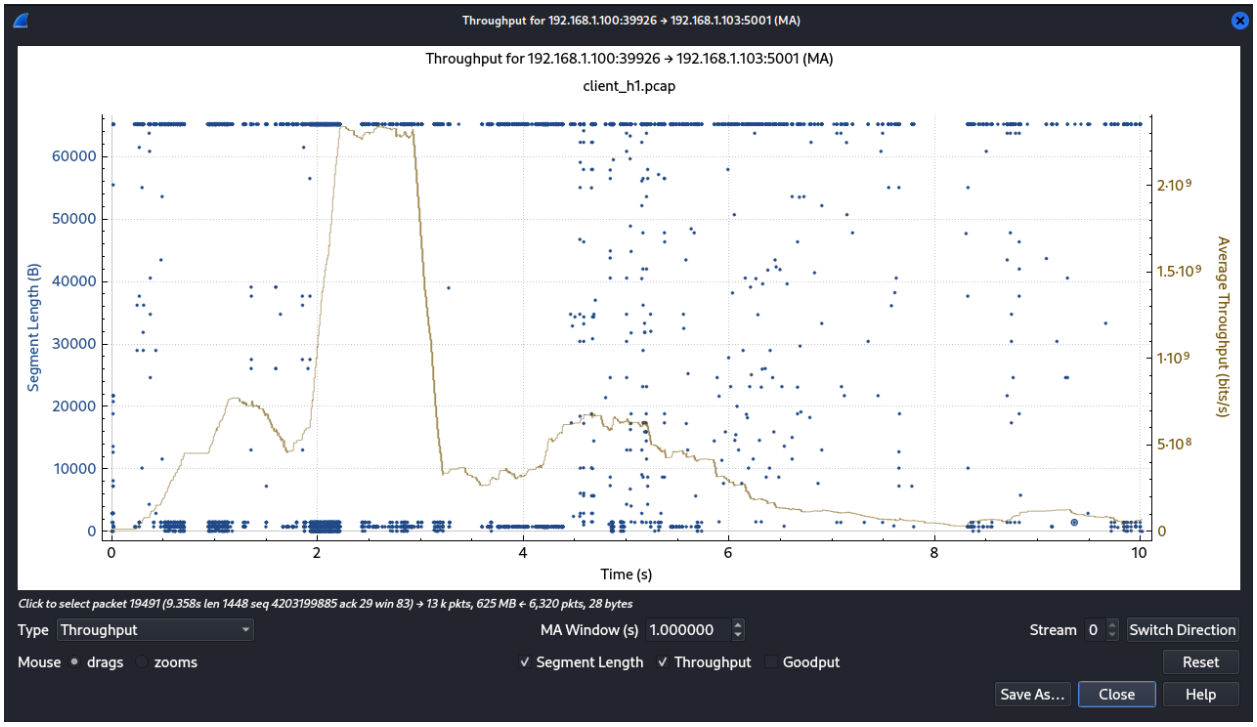


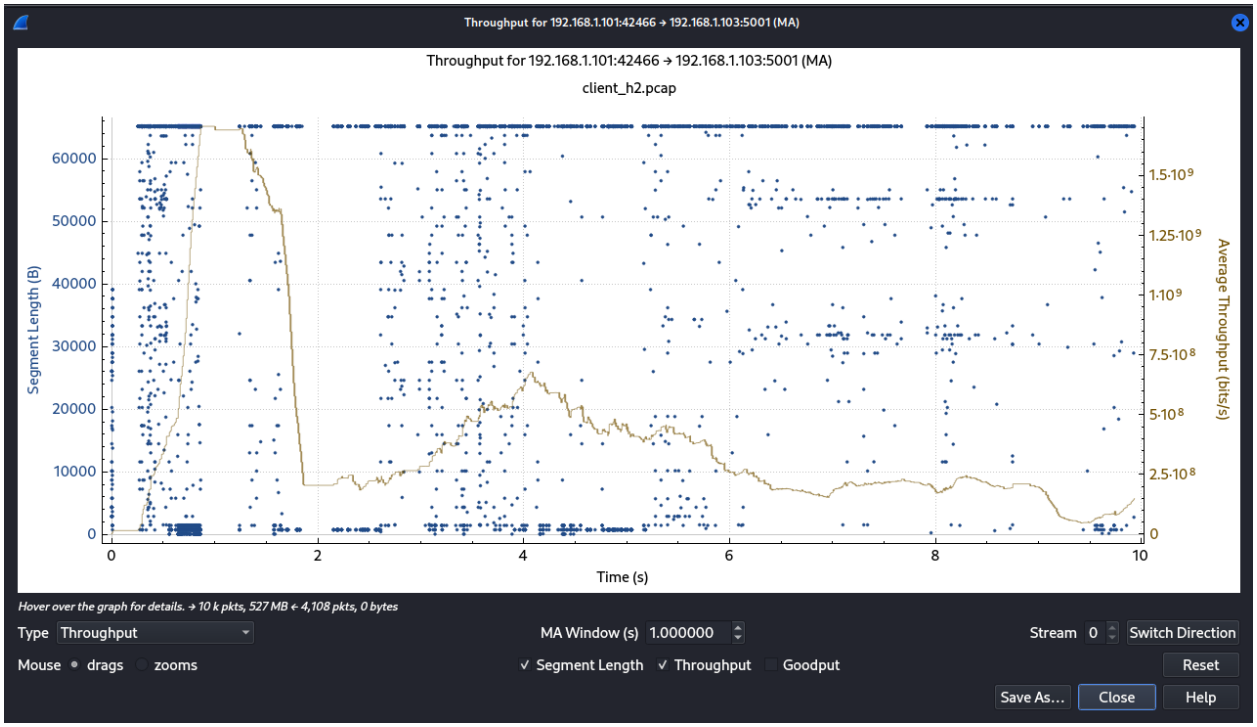

Throughput for 192.168.1.101:53556 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.101:53556 → 192.168.1.103:5001 (MA)
client_h2.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 6,559 pkts, 326 MB ← 3,175 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

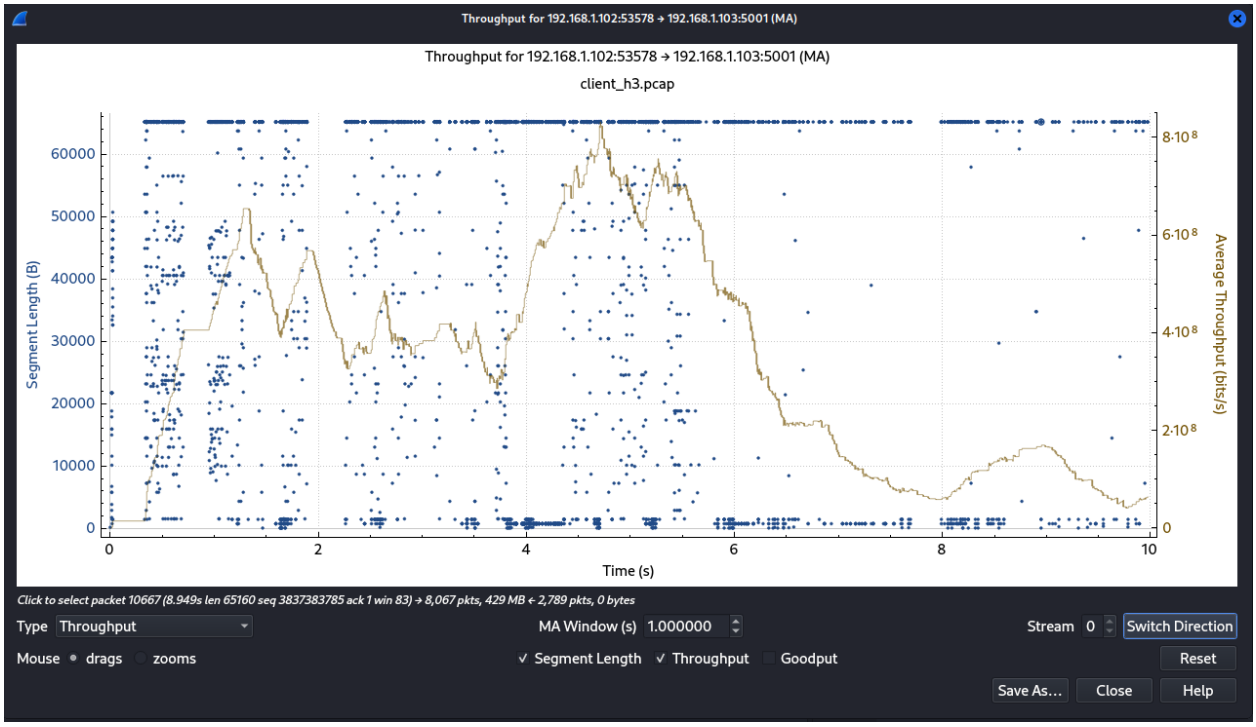

Throughput for 192.168.1.102:48382 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.102:48382 → 192.168.1.103:5001 (MA)
client_h3.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 8,307 pkts, 374 MB ← 6,918 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

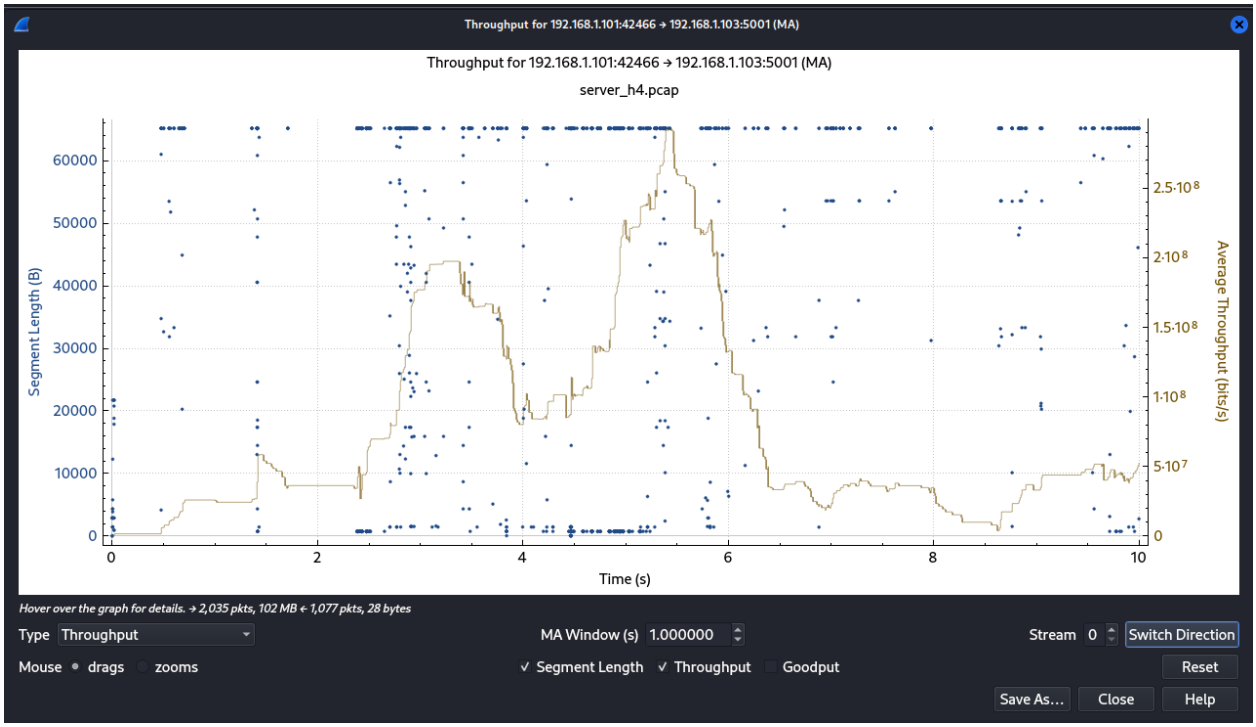

Throughput for 192.168.1.100:42856 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.100:42856 → 192.168.1.103:5001 (MA)
server_h4.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 6,440 pkts, 288 MB ← 3,247 pkts, 28 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As...
Close
Help

b. Vegas

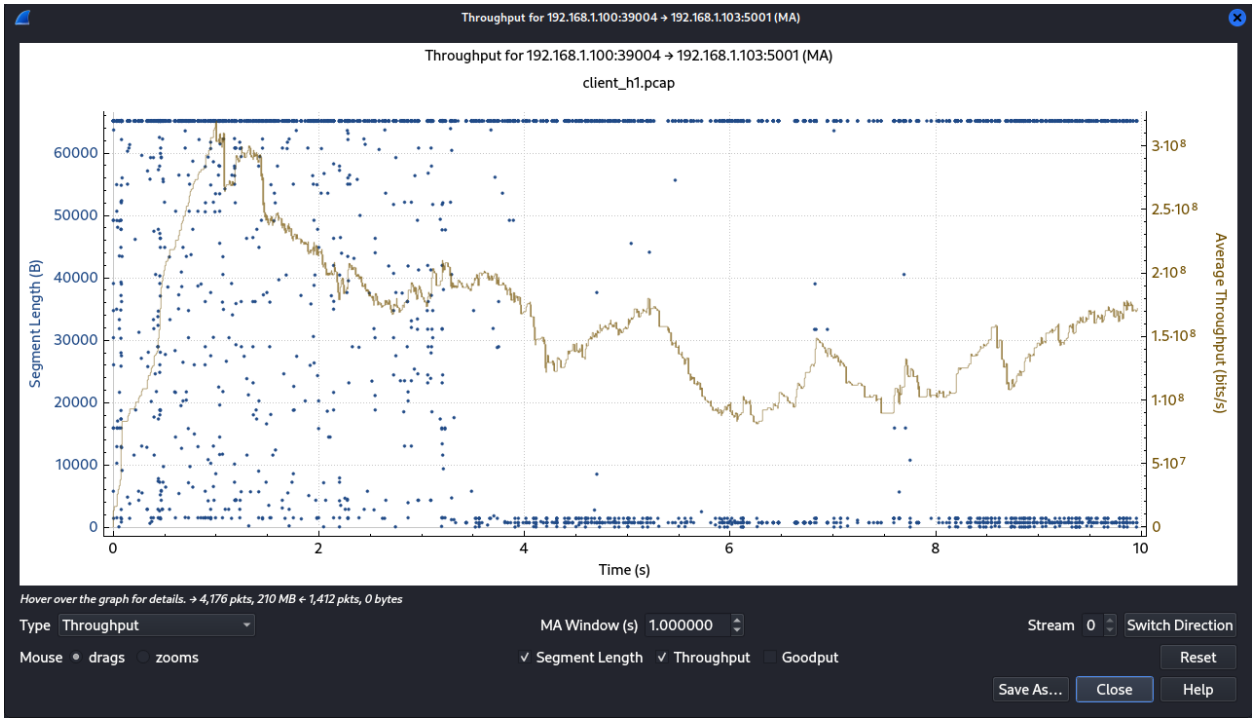

Throughput for 192.168.1.100:39926 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.100:39926 → 192.168.1.103:5001 (MA)
client_h1.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Click to select packet 19491 (9.358s len 1448 seq 4203199885 ack 29 win 83) → 13 k pkts, 625 MB ← 6,320 pkts, 28 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

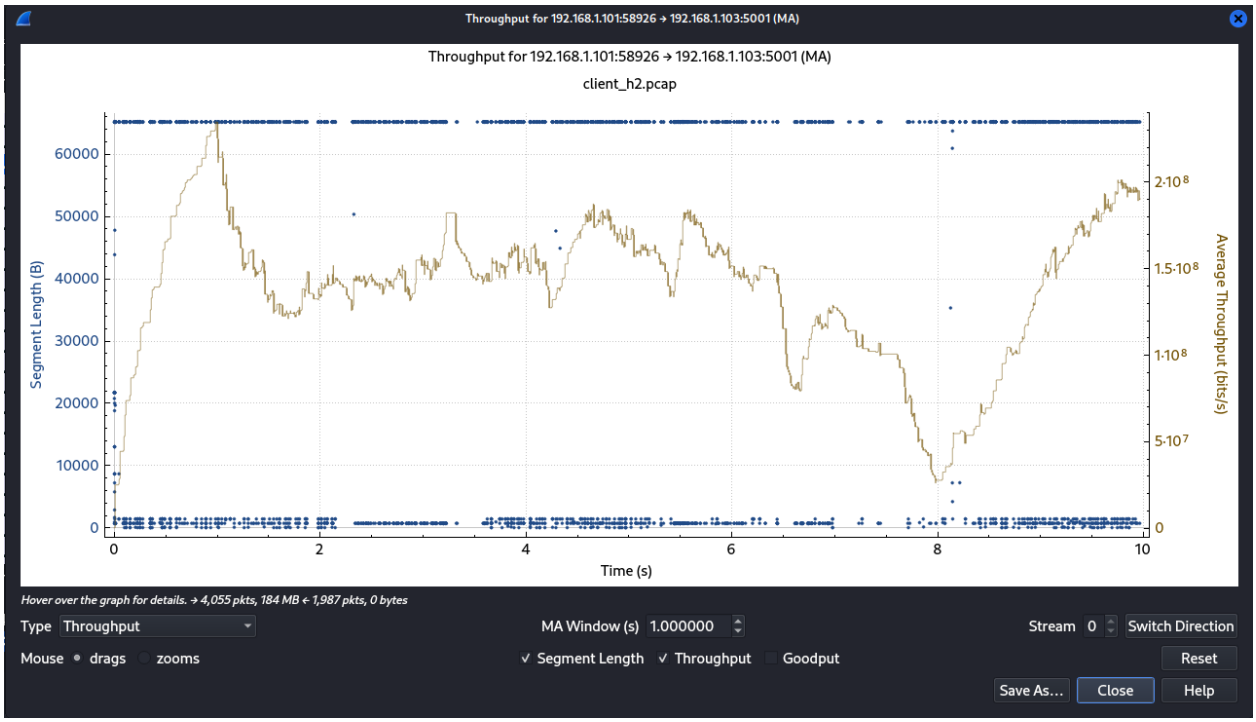

Throughput for 192.168.1.101:42466 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.101:42466 → 192.168.1.103:5001 (MA)
client_h2.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 10 k pkts, 527 MB ← 4,108 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

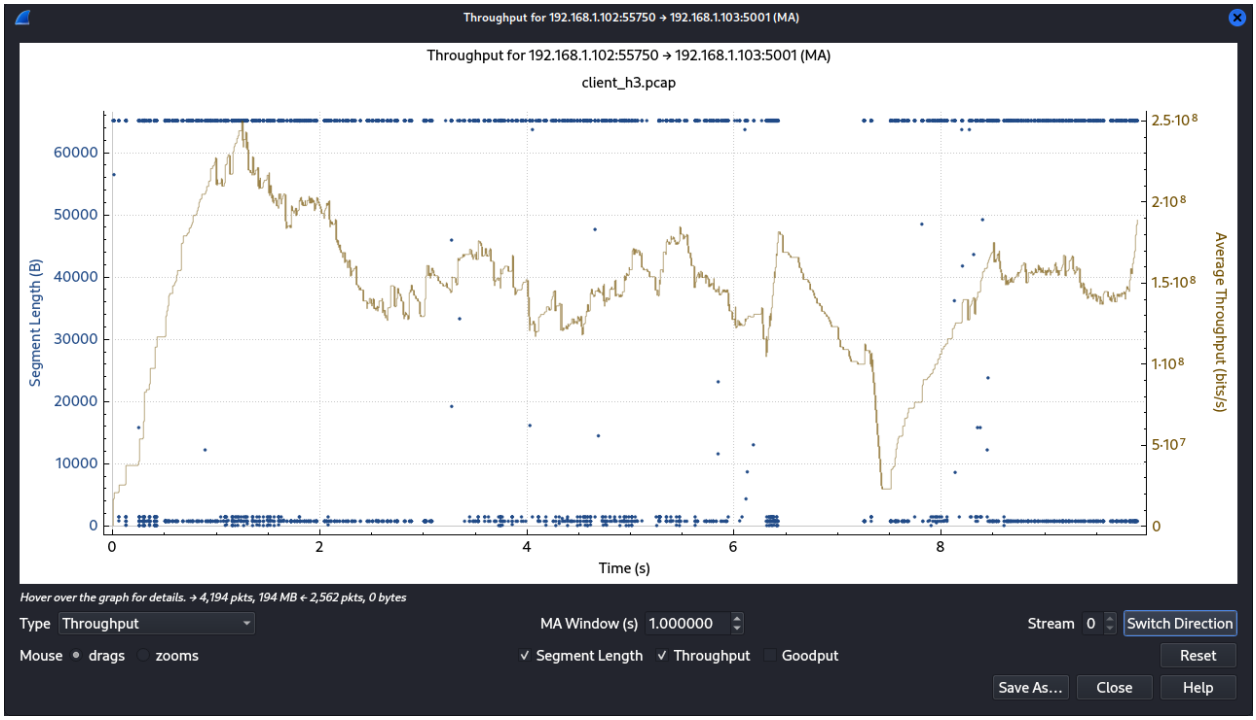

Throughput for 192.168.1.102:53578 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.102:53578 → 192.168.1.103:5001 (MA)
client_h3.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Click to select packet 10667 (8.949s len 65160 seq 3837383785 ack 1 win 83) → 8,067 pkts, 429 MB ← 2,789 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

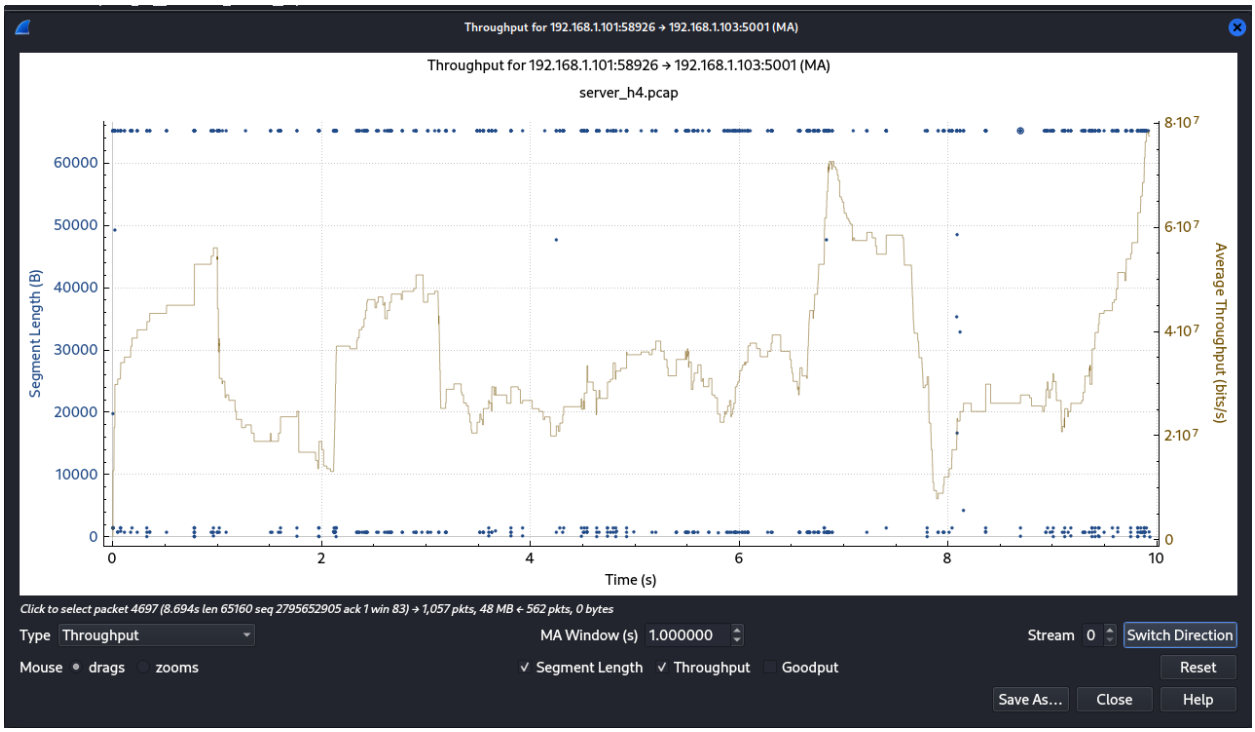

Throughput for 192.168.1.101:42466 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.101:42466 → 192.168.1.103:5001 (MA)
server_h4.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 2,035 pkts, 102 MB ← 1,077 pkts, 28 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

c. Cubic

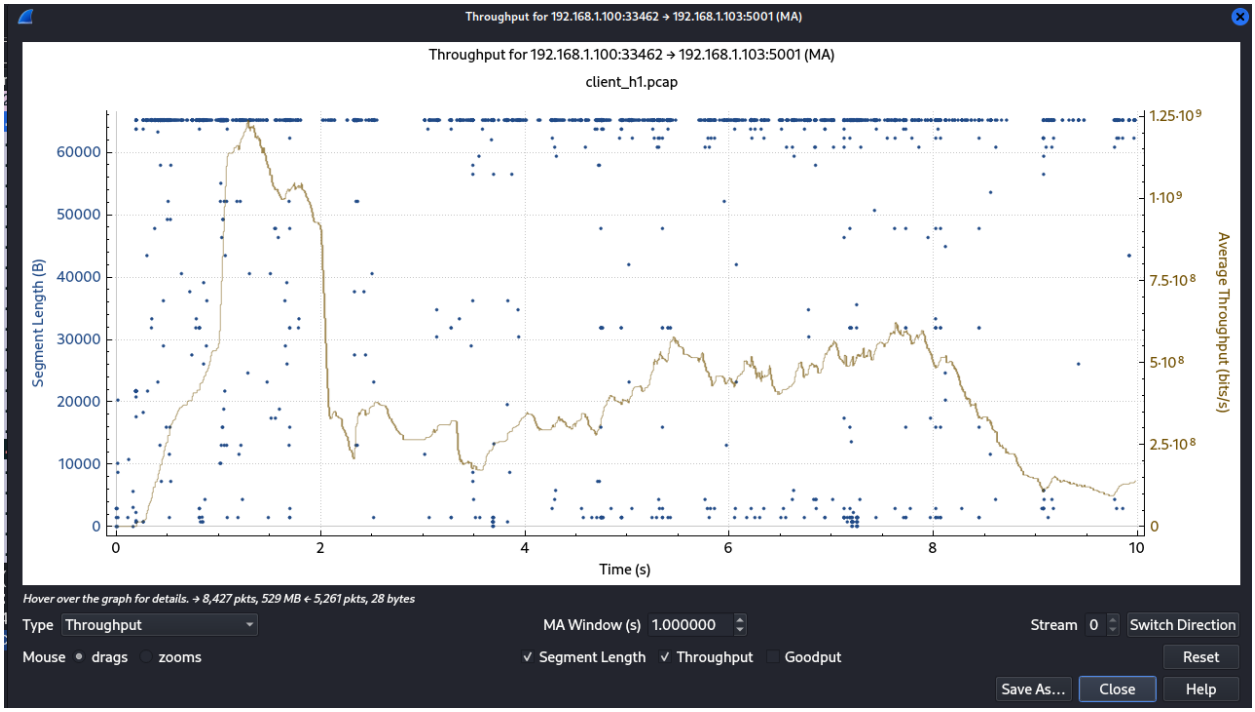

Throughput for 192.168.1.100:39004 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.100:39004 → 192.168.1.103:5001 (MA)
client_h1.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 4,176 pkts, 210 MB ← 1,412 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

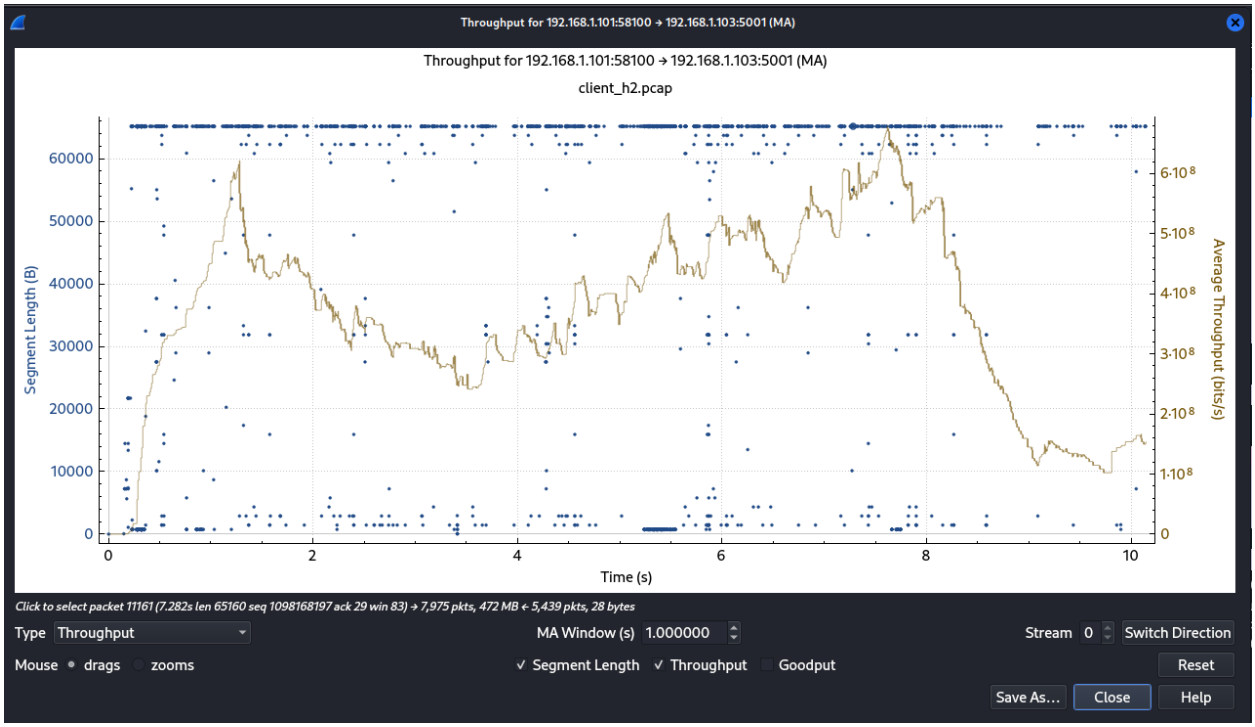

Throughput for 192.168.1.101:58926 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.101:58926 → 192.168.1.103:5001 (MA)
client_h2.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 4,055 pkts, 184 MB ← 1,987 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

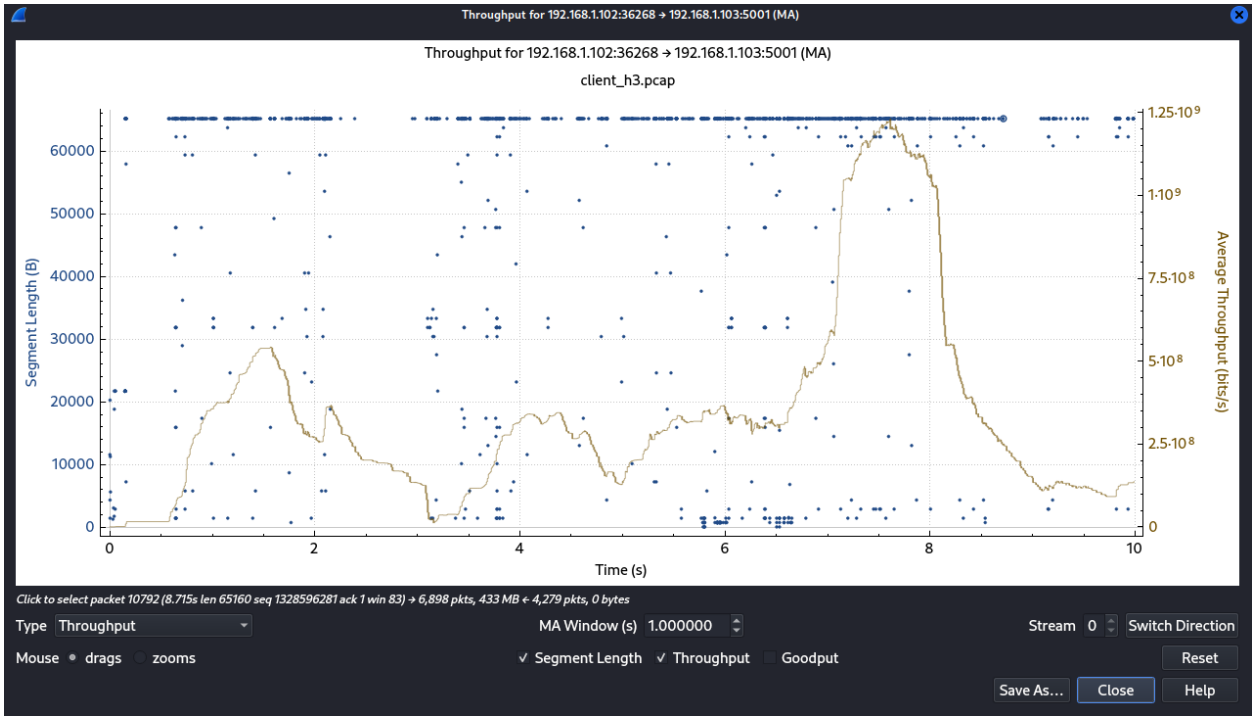

Throughput for 192.168.1.102:55750 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.102:55750 → 192.168.1.103:5001 (MA)
client_h3.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 4,194 pkts, 194 MB ← 2,562 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

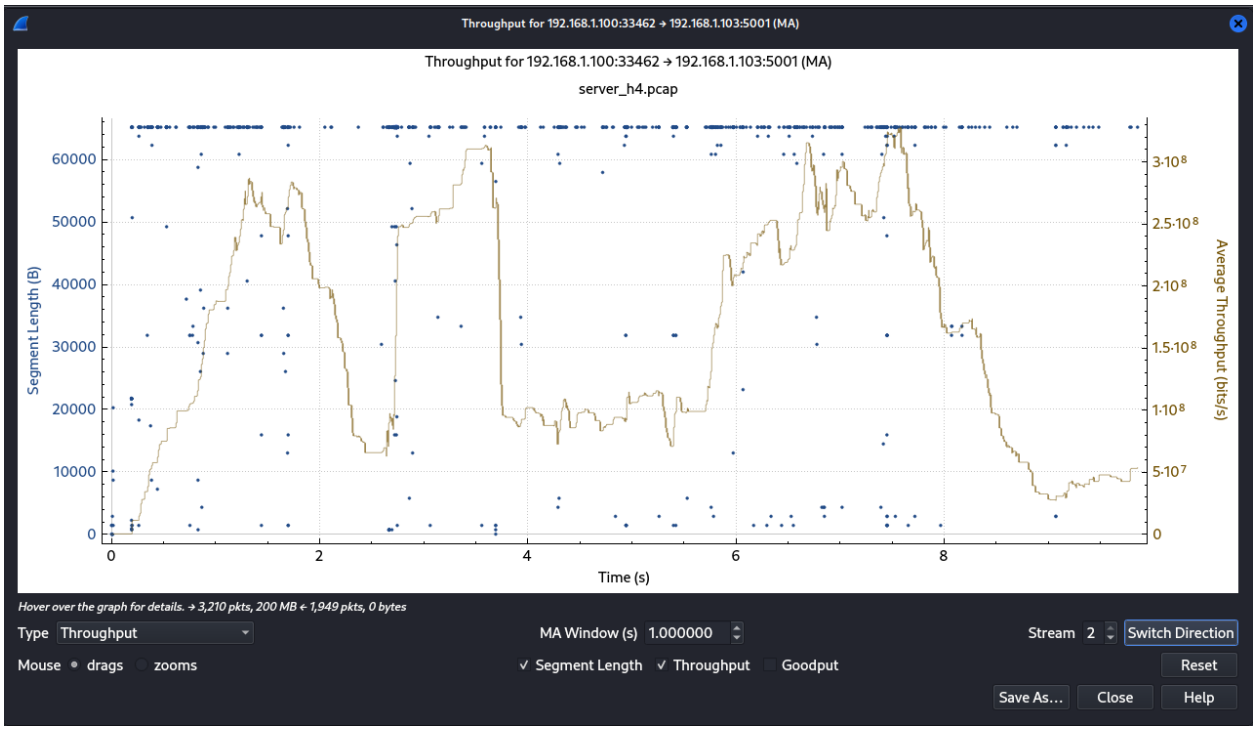

Throughput for 192.168.1.101:58926 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.101:58926 → 192.168.1.103:5001 (MA)
server_h4.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Click to select packet 4697 (8.694s len 65160 seq 2795652905 ack 1 win 83) → 1,057 pkts, 48 MB ← 562 pkts, 0 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length
Throughput
Goodput
Reset
Save As…
Close
Help

d. bbr


Throughput for 192.168.1.100:33462 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.100:33462 → 192.168.1.103:5001 (MA)
client_h1.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Hover over the graph for details. → 8,427 pkts, 529 MB ← 5,261 pkts, 28 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Save As… Close Help


Throughput for 192.168.1.101:58100 → 192.168.1.103:5001 (MA)
Throughput for 192.168.1.101:58100 → 192.168.1.103:5001 (MA)
client_h2.pcap
Segment Length (B)
Average Throughput (bits/s)
Time (s)
Click to select packet 11161 (7.282s len 65160 seq 1098168197 ack 29 win 83) → 7,975 pkts, 472 MB ← 5,439 pkts, 28 bytes
Type Throughput
MA Window (s) 1.000000
Stream 0
Switch Direction
Mouse drags zooms
Segment Length Throughput Goodput
Reset
Save As… Close Help

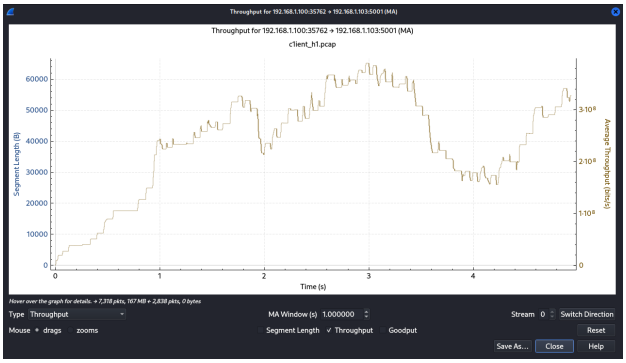
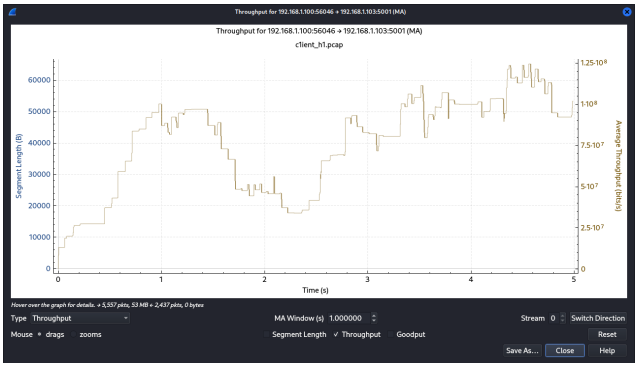
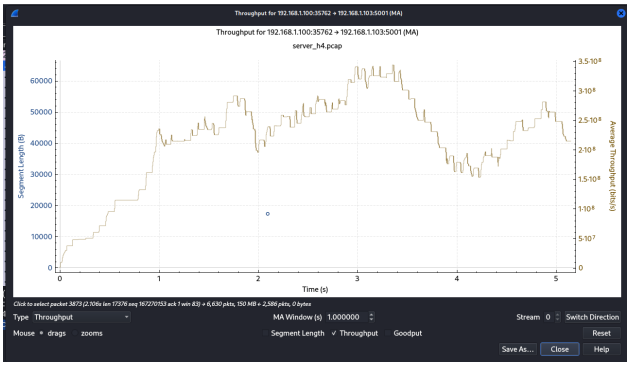
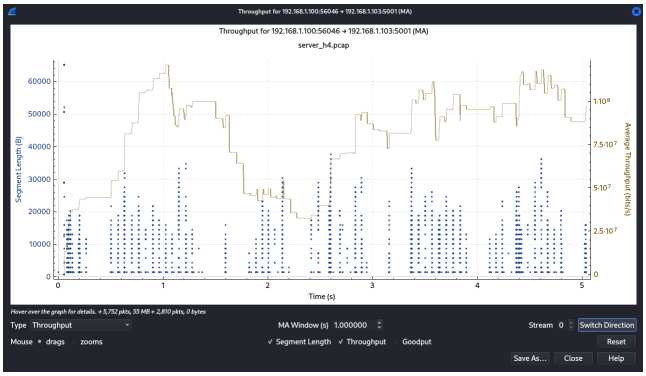




Reason: These plots are very similar to the plots as shown in part b. However here we can clearly see that all the three clients cooperate with each other to reduce their window size and roughly have the same throughput. This emphasizes the Fairness property of TCP. We can also observe that TCP Cubic and BBR have significantly higher throughputs and better fairness compared to TCP Reno and TCP Vegas.

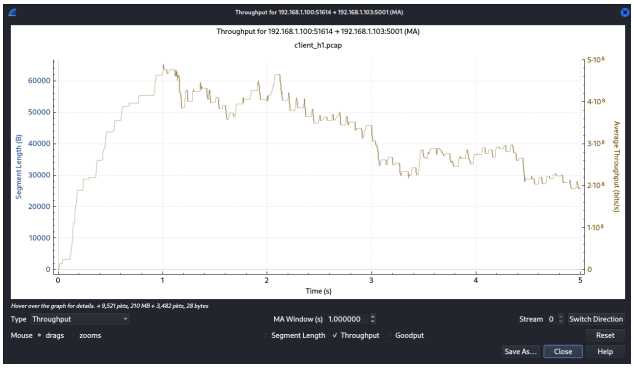
Configure the link loss parameter of the middle link (s1 - s2)

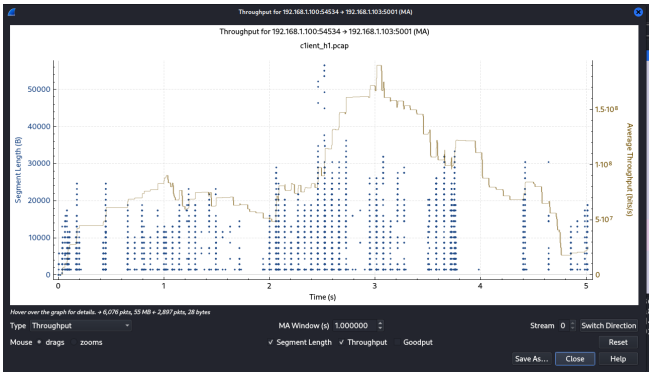
a. Reno

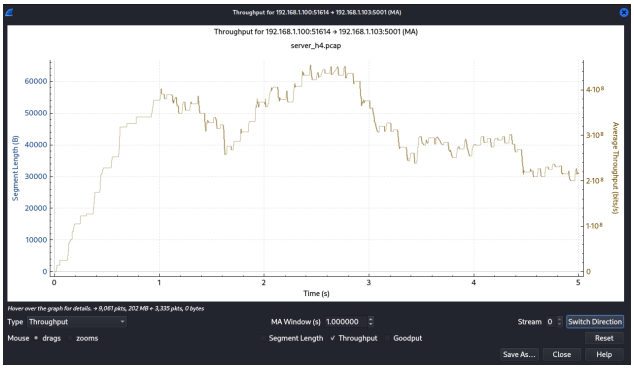
1%loss	3%loss
	
	

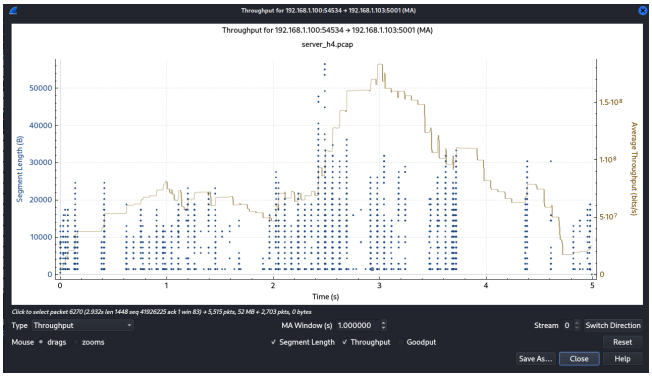
b. Vegas

1%loss	3%loss

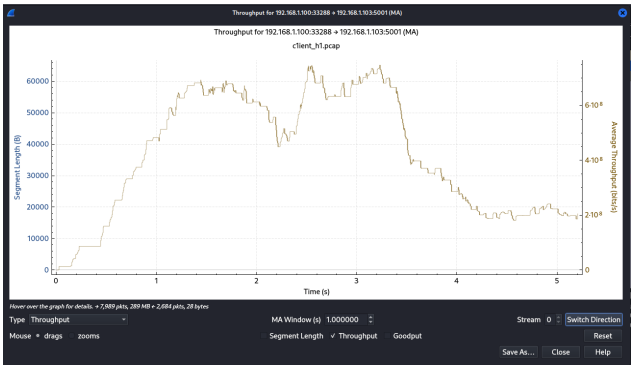
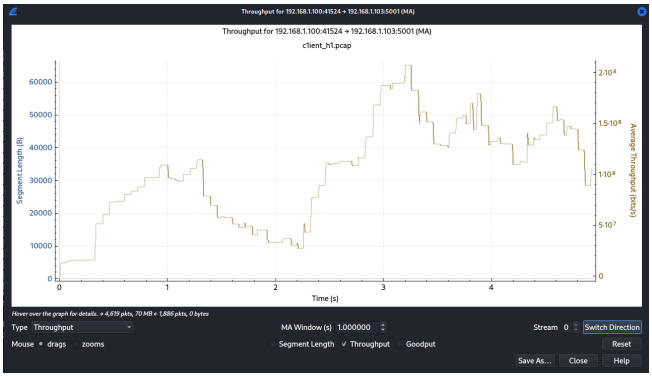


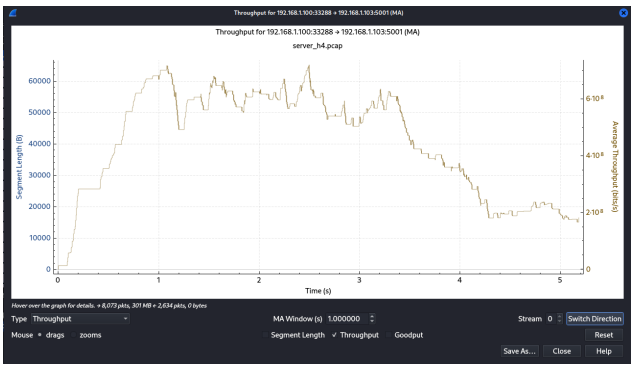


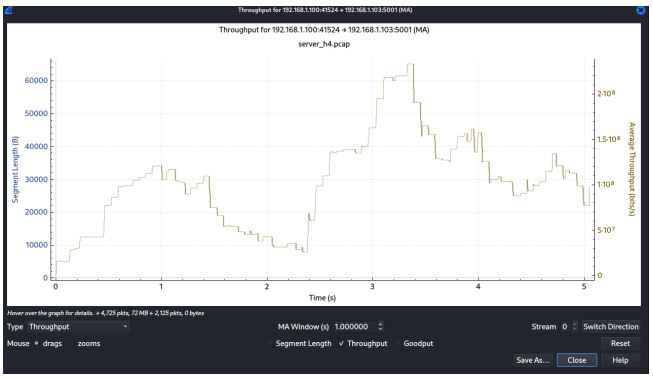




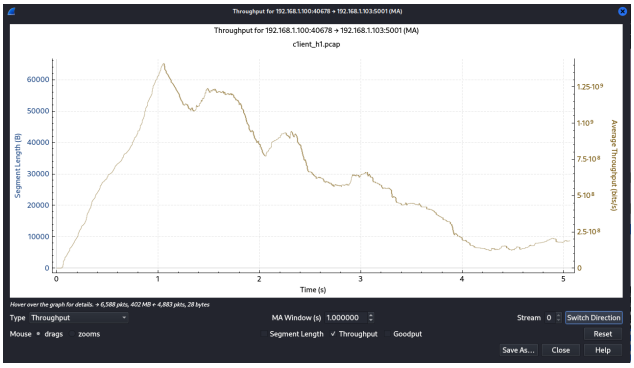
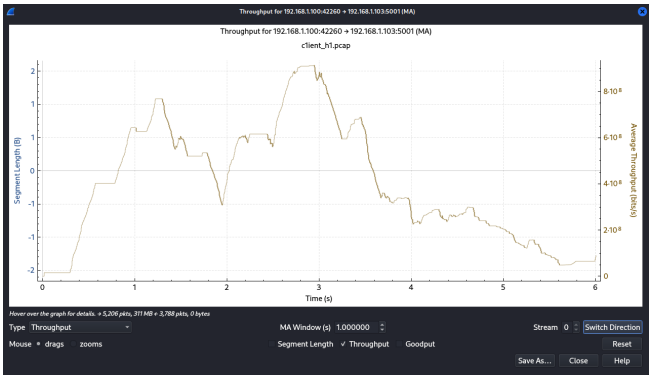
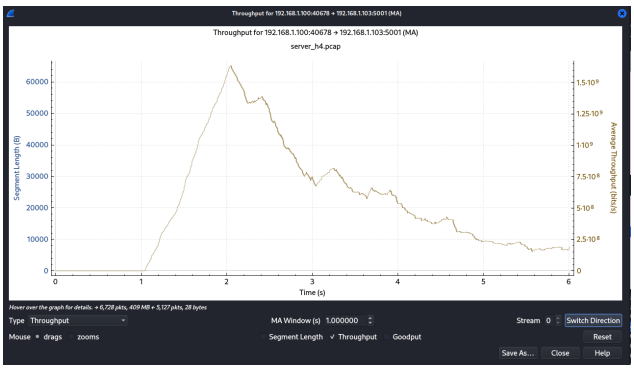
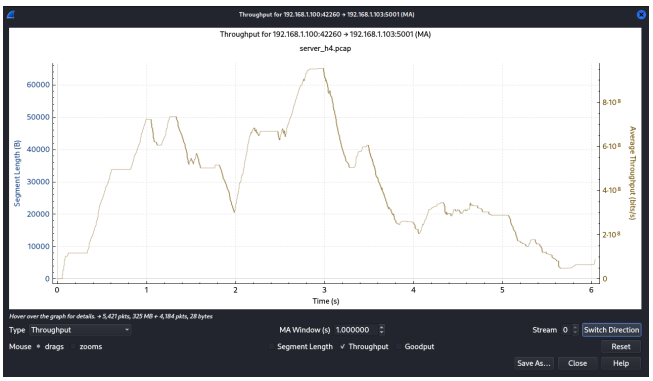
c. Cubic

1%loss	3%loss
	





d. <u>bbr</u>

<u>1%loss</u>	<u>3%loss</u>
	
	

The comparisons done through the plots give us a good idea about how the loss rate in s1-s2 link affects others.

References:

1) https://iperf.fr/iperf-doc.php
2) https://github.com/mininet/mininet/blob/master/examples/linuxrouter.py