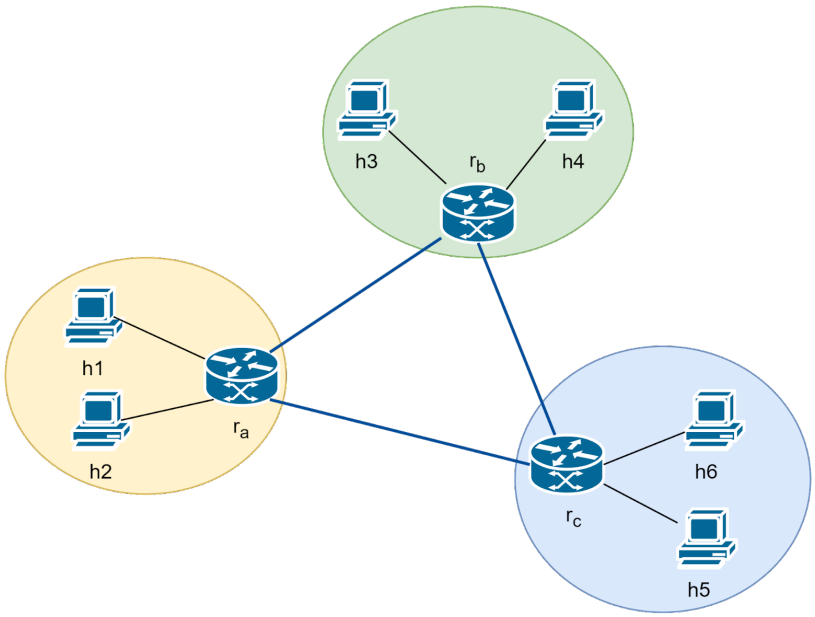
Assignment-2

Ayush Chaudhari **20110042**
Zeeshan Snehil Bhagat **20110242**

Note: Read initial setup instructions from the git repository. Also, if at any point code does not run or takes too long, make sure to check the debugging section in github.

Part I: Implement the routing functionality in mininet. (40 points)

1. Implement the below network topology in mininet. Here, the h1, h2, h3, h4, h5, and h6 are hosts, and r_a, r_b, and r_c are the routers connected to each other. You are required to implement the given network topology in mininet (using mininet's python API).



Note: As the mininet doesn't support routers natively, you can implement routers as a combination of a switch and a host. Refer to this mininet example as to how to implement a simple router.

The network is supposed to be divided into three subnets connected by the routers. Thus, you will need to assign the IP addresses to the hosts and routers/switches appropriately, maintaining the isolation.

a. **Implementation** of the custom topology using mininet. ***(10 pts)*** (Every host should be able to send packets to every other host. You are to submit the code, as well as provide screenshots as proof of its working)

Answer:

- Open terminal at the point where the program files are present.
- Run "sudo python q1_a.py" [give the password as well for sudo authorization]
- Run "pingall"

```
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)
```

As you can see, all the six hosts, h1-h6 can communicate with each other.

b. **Observations**: Capture and show the wireshark/tcpdump (packets) for the route setup on any one of the routers.***(10 pts)***

Answer:

- Run “ra wireshark &” [you can use rb or rc for routers as well]
- Select any interface [any (highly preferred), or ra-eth…]
- Now run pingall in the terminal and observe wireshark

	Time	Source	Destination	Protocol	Length	Info
1	0.000000000	172.16.0.2	172.16.0.3	ICMP	100	Echo (ping) request id=0x36a3, seq=1/256, ttl=64 (no response found!)
2	0.018764657	172.16.0.2	172.16.1.2	ICMP	100	Echo (ping) request id=0x1f4c, seq=1/256, ttl=64 (no response found!)
3	0.018781564	172.16.0.2	172.16.1.2	ICMP	100	Echo (ping) request id=0x1f4c, seq=1/256, ttl=63 (reply in 4)
4	0.020557855	172.16.1.2	172.16.0.2	ICMP	100	Echo (ping) reply id=0x1f4c, seq=1/256, ttl=63 (request in 3)
5	0.020573593	172.16.1.2	172.16.0.2	ICMP	100	Echo (ping) reply id=0x1f4c, seq=1/256, ttl=62
6	0.027229177	172.16.0.2	172.16.1.3	ICMP	100	Echo (ping) request id=0xa181, seq=1/256, ttl=64 (no response found!)
7	0.027241185	172.16.0.2	172.16.1.3	ICMP	100	Echo (ping) request id=0xa181, seq=1/256, ttl=63 (reply in 8)
8	0.027854128	172.16.1.3	172.16.0.2	ICMP	100	Echo (ping) reply id=0xa181, seq=1/256, ttl=63 (request in 7)
9	0.027856382	172.16.1.3	172.16.0.2	ICMP	100	Echo (ping) reply id=0xa181, seq=1/256, ttl=62
10	0.033074444	172.16.0.2	172.16.2.2	ICMP	100	Echo (ping) request id=0x3bb3, seq=1/256, ttl=64 (no response found!)
11	0.033085160	172.16.0.2	172.16.2.2	ICMP	100	Echo (ping) request id=0x3bb3, seq=1/256, ttl=63 (reply in 12)
12	0.033675016	172.16.2.2	172.16.0.2	ICMP	100	Echo (ping) reply id=0x3bb3, seq=1/256, ttl=63 (request in 11)
13	0.033677193	172.16.2.2	172.16.0.2	ICMP	100	Echo (ping) reply id=0x3bb3, seq=1/256, ttl=62
14	0.039313868	172.16.0.2	172.16.2.3	ICMP	100	Echo (ping) request id=0xe8d4, seq=1/256, ttl=64 (no response found!)
15	0.039324356	172.16.0.2	172.16.2.3	ICMP	100	Echo (ping) request id=0xe8d4, seq=1/256, ttl=63 (reply in 16)
16	0.040033532	172.16.2.3	172.16.0.2	ICMP	100	Echo (ping) reply id=0xe8d4, seq=1/256, ttl=63 (request in 15)
17	0.040035831	172.16.2.3	172.16.0.2	ICMP	100	Echo (ping) reply id=0xe8d4, seq=1/256, ttl=62
18	0.055315328	172.16.0.2	172.16.0.1	ICMP	100	Echo (ping) request id=0x87f9, seq=1/256, ttl=64 (reply in 19)
19	0.055336373	172.16.0.1	172.16.0.2	ICMP	100	Echo (ping) reply id=0x87f9, seq=1/256, ttl=64 (request in 18)
20	0.063424793	172.16.0.2	172.16.1.1	ICMP	100	Echo (ping) request id=0xa5af, seq=1/256, ttl=64 (no response found!)

Here you can observe all the packets traversing through router ra.

c. Vary the default routing and measure the latency difference. (i.e. the default route for a packet from h1 to h6 would be h1 -> r_a -> r_c -> h6. Try to vary the path so that it takes the path h1-> r_a -> r_b -> r_c -> h6. Provide the screenshots as proof for the latency difference, both ping as well as iperf) ***(10 pts)***

Answer:

- Run “h1 ping -c 10 h6” [for ping latency]

```
mininet> h1 ping -c 30 h6
PING 172.16.2.3 (172.16.2.3) 56(84) bytes of data.
64 bytes from 172.16.2.3: icmp_seq=1 ttl=62 time=1.94 ms
64 bytes from 172.16.2.3: icmp_seq=2 ttl=62 time=0.109 ms
64 bytes from 172.16.2.3: icmp_seq=3 ttl=62 time=0.113 ms
64 bytes from 172.16.2.3: icmp_seq=4 ttl=62 time=0.098 ms
64 bytes from 172.16.2.3: icmp_seq=5 ttl=62 time=0.106 ms
```

```
── 172.16.2.3 ping statistics ──
30 packets transmitted, 30 received, 0% packet loss, time 29644ms
rtt min/avg/max/mdev = 0.058/0.200/1.937/0.353 ms
```

- Run “h6 iperf -s -i 1 &”
- Run “h1 iperf -c h6 -t 10 -i 1”

```
mininet> h6 iperf -s -i 1 &
------------------------------------------------------------
Server listening on TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
mininet> h1 iperf -c h6 -t 10 -i 1
------------------------------------------------------------
Client connecting to 172.16.2.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 172.16.0.2 port 35910 connected with 172.16.2.3 port 5001 (icwnd/mss/irtt=14/1448/10728)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  3.19 GBytes  27.4 Gbits/sec
[  1] 1.0000-2.0000 sec  3.25 GBytes  27.9 Gbits/sec
[  1] 2.0000-3.0000 sec  3.23 GBytes  27.8 Gbits/sec
[  1] 3.0000-4.0000 sec  3.19 GBytes  27.4 Gbits/sec
[  1] 4.0000-5.0000 sec  3.16 GBytes  27.1 Gbits/sec
[  1] 5.0000-6.0000 sec  2.98 GBytes  25.6 Gbits/sec
[  1] 6.0000-7.0000 sec  3.07 GBytes  26.4 Gbits/sec
[  1] 7.0000-8.0000 sec  3.22 GBytes  27.7 Gbits/sec
[  1] 8.0000-9.0000 sec  3.31 GBytes  28.4 Gbits/sec
[  1] 9.0000-10.0000 sec  3.29 GBytes  28.3 Gbits/sec
[  1] 0.0000-10.0070 sec  31.9 GBytes  27.4 Gbits/sec
mininet>
```

- Ctrl+D [If the previous program is running on mininet]
- Now, run “sudo python q1_c.py”
- Now repeat the steps as done for q1_a.py

```
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 X X rb X
rb → h1 h2 h3 h4 h5 h6 ra rc
rc → X X h3 h4 h5 h6 X rb
*** Results: 8% dropped (66/72 received)
```

```
mininet> h1 ping -c 30 h6
PING 172.16.2.3 (172.16.2.3) 56(84) bytes of data.
64 bytes from 172.16.2.3: icmp_seq=1 ttl=61 time=7.45 ms
64 bytes from 172.16.2.3: icmp_seq=2 ttl=61 time=3.03 ms
64 bytes from 172.16.2.3: icmp_seq=3 ttl=61 time=0.117 ms
64 bytes from 172.16.2.3: icmp_seq=4 ttl=61 time=0.109 ms
64 bytes from 172.16.2.3: icmp_seq=5 ttl=61 time=0.106 ms
64 bytes from 172.16.2.3: icmp_seq=6 ttl=61 time=0.079 ms
```

```
── 172.16.2.3 ping statistics ──
30 packets transmitted, 30 received, 0% packet loss, time 30710ms
rtt min/avg/max/mdev = 0.062/0.468/7.449/1.399 ms
```

```
mininet> h6 iperf -s -i 1 &
mininet> h1 iperf -c h6 -t 10 -i 1
------------------------------------------------------------
Client connecting to 172.16.2.3, TCP port 5001
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 172.16.0.2 port 34432 connected with 172.16.2.3 port 5001 (icwnd/mss/irtt=14/1448/6169)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  2.89 GBytes  24.8 Gbits/sec
[  1] 1.0000-2.0000 sec  3.10 GBytes  26.7 Gbits/sec
[  1] 2.0000-3.0000 sec  3.12 GBytes  26.8 Gbits/sec
[  1] 3.0000-4.0000 sec  3.09 GBytes  26.5 Gbits/sec
[  1] 4.0000-5.0000 sec  3.09 GBytes  26.6 Gbits/sec
[  1] 5.0000-6.0000 sec  2.94 GBytes  25.2 Gbits/sec
[  1] 6.0000-7.0000 sec  3.11 GBytes  26.7 Gbits/sec
[  1] 7.0000-8.0000 sec  3.07 GBytes  26.4 Gbits/sec
[  1] 8.0000-9.0000 sec  3.06 GBytes  26.3 Gbits/sec
[  1] 9.0000-10.0000 sec  3.02 GBytes  25.9 Gbits/sec
[  1] 0.0000-10.0116 sec  30.5 GBytes  26.2 Gbits/sec
```

d. Dump the routing tables for all the routers for questions a & c. ***(10 pts)***

Answer:

Part a:

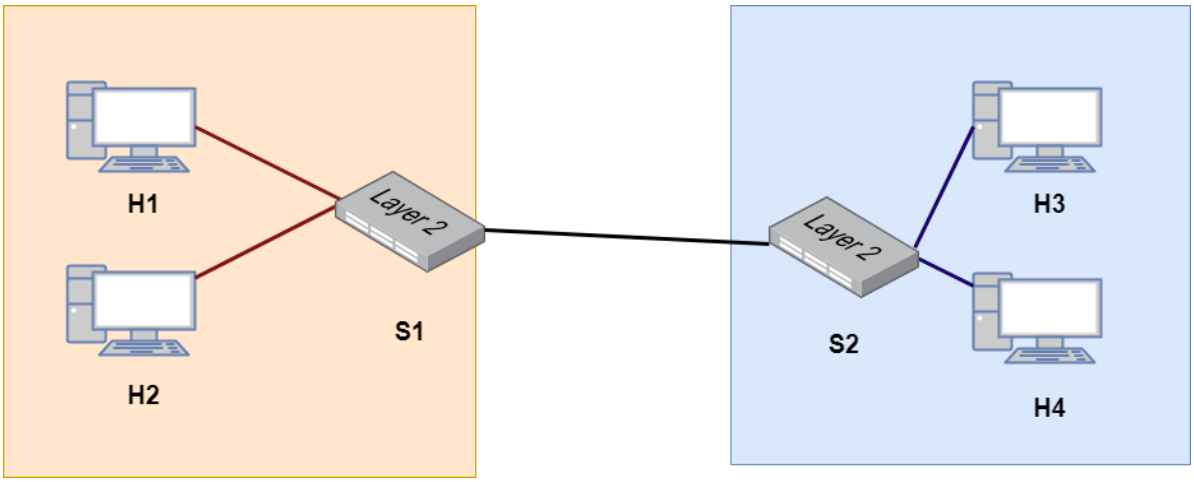
```
*** Routing Tables on Routers:
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      0.0.0.0         255.255.255.0   U     0      0        0 ra-eth1
172.16.1.0      172.16.3.2      255.255.255.0   UG    0      0        0 ra-eth2
172.16.2.0      172.16.5.1      255.255.255.0   UG    0      0        0 ra-eth3
172.16.3.0      0.0.0.0         255.255.255.0   U     0      0        0 ra-eth2
172.16.5.0      0.0.0.0         255.255.255.0   U     0      0        0 ra-eth3
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      172.16.3.1      255.255.255.0   UG    0      0        0 rb-eth2
172.16.1.0      0.0.0.0         255.255.255.0   U     0      0        0 rb-eth1
172.16.2.0      172.16.4.2      255.255.255.0   UG    0      0        0 rb-eth3
172.16.3.0      0.0.0.0         255.255.255.0   U     0      0        0 rb-eth2
172.16.4.0      0.0.0.0         255.255.255.0   U     0      0        0 rb-eth3
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      172.16.5.2      255.255.255.0   UG    0      0        0 rc-eth3
172.16.1.0      172.16.4.1      255.255.255.0   UG    0      0        0 rc-eth2
172.16.2.0      0.0.0.0         255.255.255.0   U     0      0        0 rc-eth1
172.16.4.0      0.0.0.0         255.255.255.0   U     0      0        0 rc-eth2
172.16.5.0      0.0.0.0         255.255.255.0   U     0      0        0 rc-eth3
*** Starting CLI:
```

Part c:

```
*** Routing Tables on Routers:
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      0.0.0.0         255.255.255.0   U     0      0        0 ra-eth1
172.16.1.0      172.16.3.2      255.255.255.0   UG    0      0        0 ra-eth2
172.16.2.0      172.16.3.2      255.255.255.0   UG    0      0        0 ra-eth2
172.16.3.0      0.0.0.0         255.255.255.0   U     0      0        0 ra-eth2
172.16.5.0      0.0.0.0         255.255.255.0   U     0      0        0 ra-eth3
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      172.16.3.1      255.255.255.0   UG    0      0        0 rb-eth2
172.16.1.0      0.0.0.0         255.255.255.0   U     0      0        0 rb-eth1
172.16.2.0      172.16.4.2      255.255.255.0   UG    0      0        0 rb-eth3
172.16.3.0      0.0.0.0         255.255.255.0   U     0      0        0 rb-eth2
172.16.4.0      0.0.0.0         255.255.255.0   U     0      0        0 rb-eth3
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
172.16.0.0      172.16.4.1      255.255.255.0   UG    0      0        0 rc-eth2
172.16.1.0      172.16.4.1      255.255.255.0   UG    0      0        0 rc-eth2
172.16.2.0      0.0.0.0         255.255.255.0   U     0      0        0 rc-eth1
172.16.4.0      0.0.0.0         255.255.255.0   U     0      0        0 rc-eth2
172.16.5.0      0.0.0.0         255.255.255.0   U     0      0        0 rc-eth3
*** Starting CLI:
```

Part II: Throughput for different congestion control schemes. (35 points)

Create a mininet topology as shown in the diagram below. H1 to H4 are four hosts connected to switches S1 and S2. Run a TCP Server that accepts data in the H4 system. H1, H2, H3 are the TCP clients that connect with the server (H4) and sends data packets.



For the congestion control schemes: Vegas, Reno, Cubic, and BBR, identify the throughput (graph) observed. You can use wireshark to identify the throughput.

You can implement a custom client-server program that generates TCP data packets

for the required analysis. Ex: a File transfer program, HTTP server or simple load generator tools like iperf.

Throughput analysis can also be done using the iPerf tool. Explore how to set up a client–server for analysis of TCP connections and configuring Congestion control schemes in the official documentation from the iPerf website: https://iperf.fr/.

a. Implementation of the mininet topology and TCP client-server program (for packet generation).

   The TCP client-server program should run with the required configuration for question (b) when `--config=b` and with required configuration for question (c) when `--config=c`

   Other configurable parameters like congestion control scheme, link loss should also be taken as command line arguments to the program. **(5 points)**

Answer:

1. Write “sudo python q2.py” at the terminal opened at the folder where the code file is present
2. Now, you may or may not add parameters related to config={b, c}, congestion={Reno, BBR, Cubic, Vegas} and loss=any value in percentage
3. Default parameters are config=b, congestion=Cubic, loss=0. Just pressing enter will run the code.
4. To add parameters, after “... q2.py “, write “--config=c” for config and similar things for others. Note that for congestion, write exactly how it has been written in 2nd point(First capital letter, then small letters)
5. All the plots for part b,c,d were generated after running the code using all the combinations of parameters.

b. Run the client on H1 and the server on H4. Report and reason the throughput over time for each congestion control scheme. **(10 points)**

Answer:

BBR (Bottleneck Bandwidth and Round-trip propagation time) is a congestion control algorithm that aims to achieve high bandwidth and low latency for TCP traffic. It does this by estimating the available bandwidth and adjusting the sending rate accordingly. BBR is also able to quickly recover from congestion, which helps to maintain high throughput even when there is network congestion.

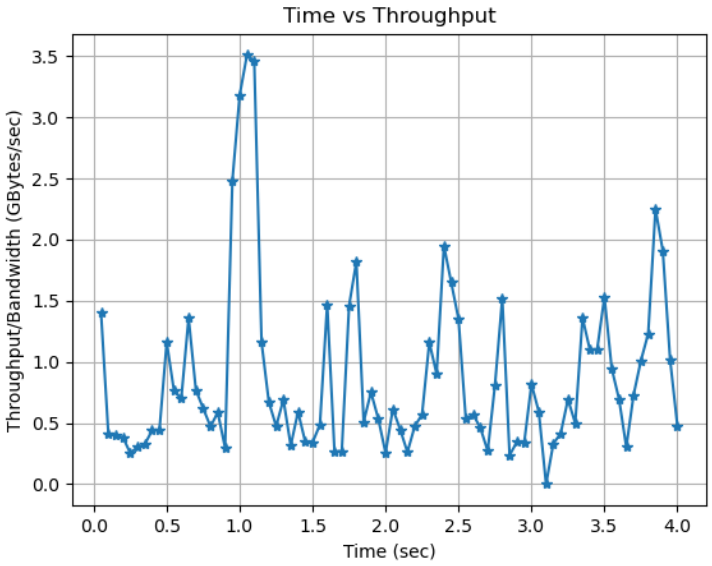
Cubic is a congestion control algorithm that is similar to Reno, but it is more conservative. This means that Cubic is less likely to cause congestion, but it also means that it may not achieve the highest possible throughput.

Reno is the default congestion control algorithm for TCP. It is a simple and effective algorithm, but it is not as efficient as BBR or Cubic.

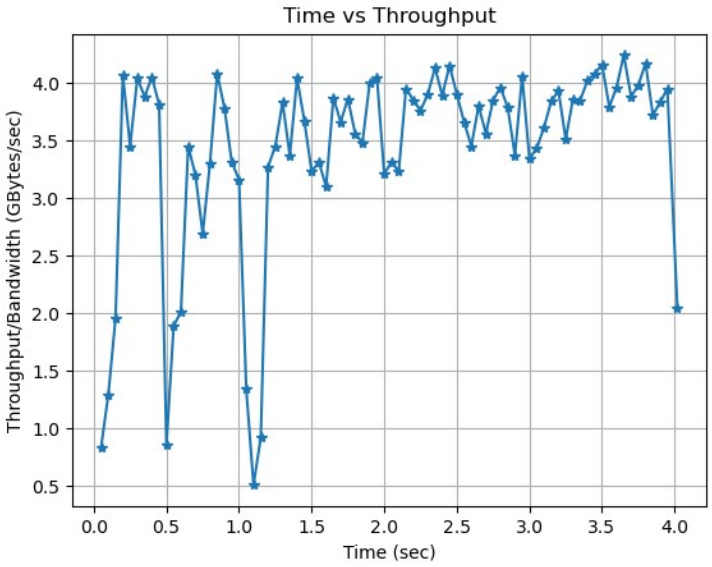
Vegas is a congestion control algorithm that is designed to be fair. This means that it tries to ensure that all flows receive equal bandwidth. Vegas does this by estimating the

available bandwidth and then dividing it evenly among all of the flows.

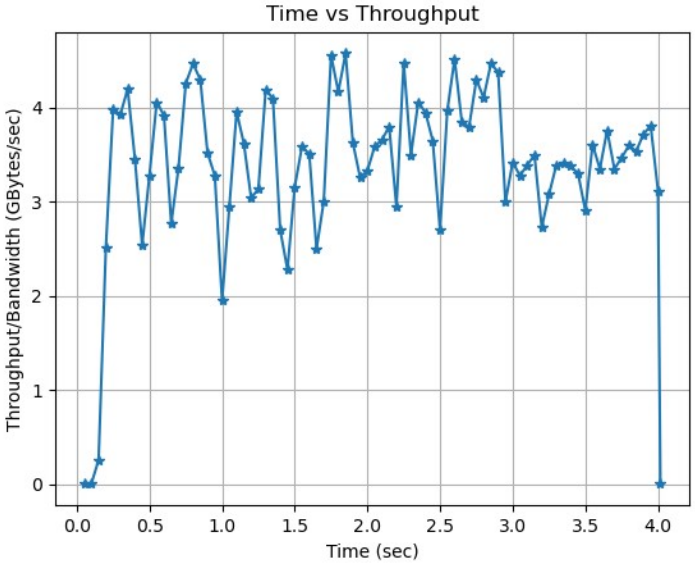
However, the plots changed every time we ran the code, sometimes very significantly, possibly due to the machine setup.
Cubic and Reno show the most consistent throughput followed by Vegas and BBR. Vegas showed sharp drops. BBR might have been unable to perform well due to the then state of the machine. BBR also showed gradual drops.



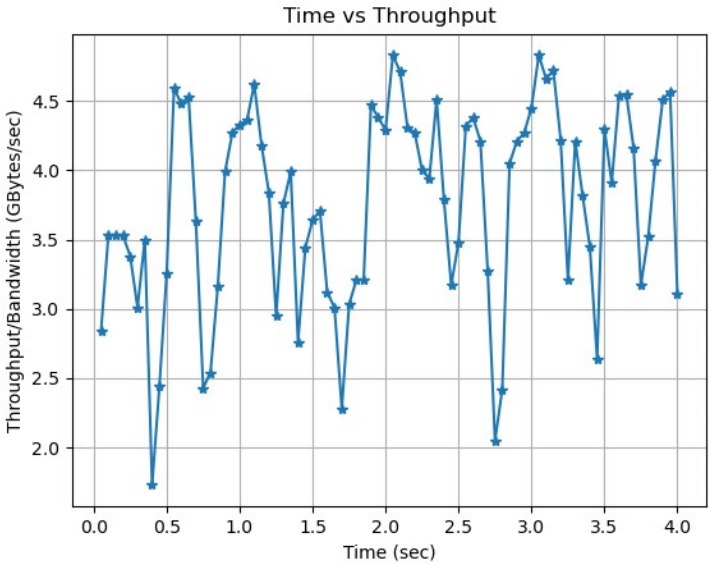
[config_b_host_1_congestion_BBR_loss_0.0]



[config_b_host_1_congestion_Cubic_loss_0.0]



[config_b_host_1_congestion_Reno_loss_0.0]

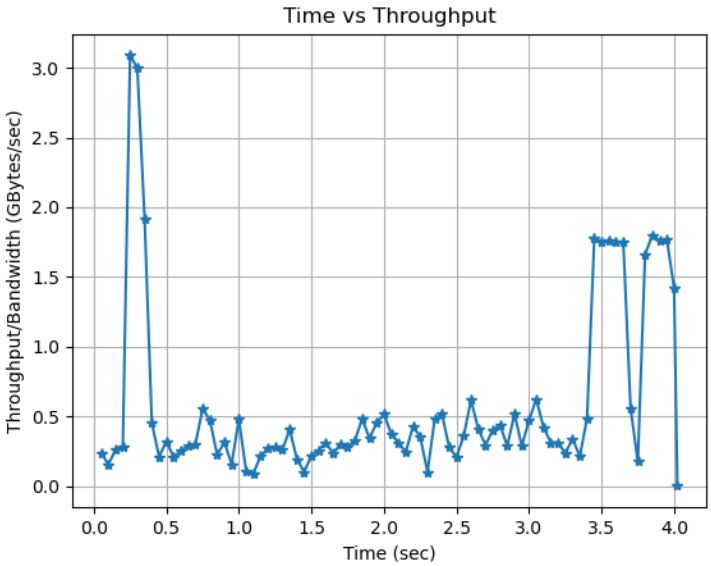


[config_b_host_1_congestion_Vegas_loss_0.0]

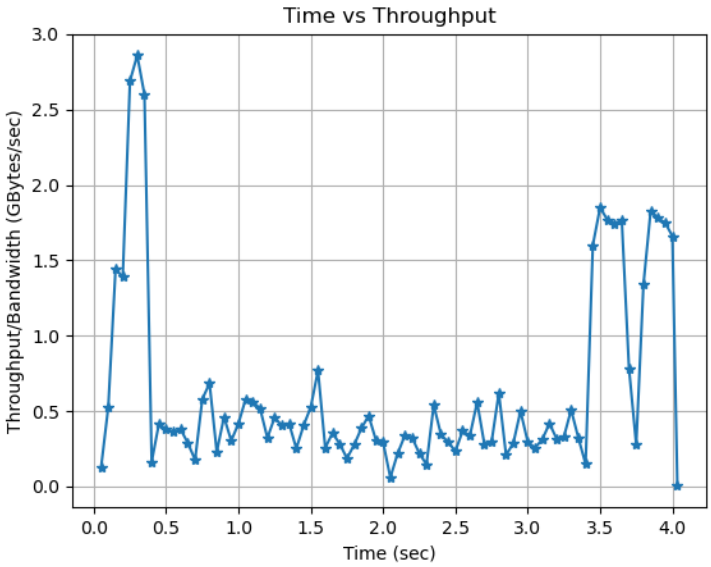
c. Run the client on H1, H2, H3 simultaneously and the server on H4. Report and reason the throughput over time for each host, each congestion control scheme. **(10 points)**

Answer:

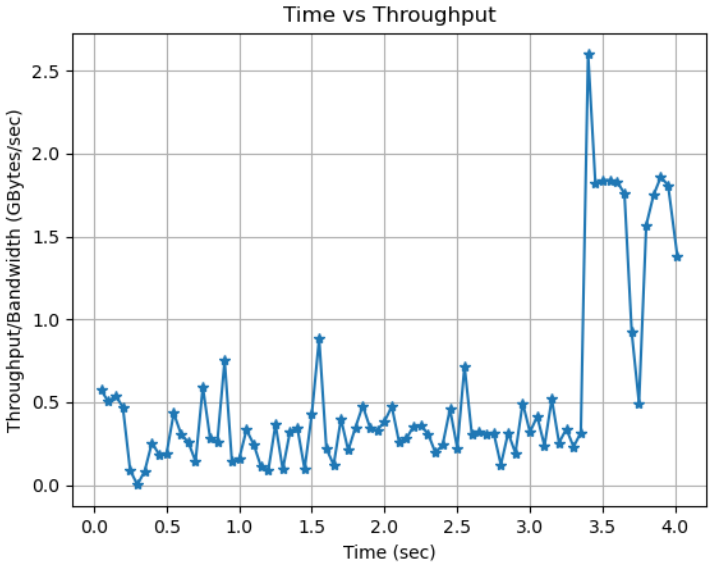
All 3 clients showed similar throughput for each congestion. There was a significant dip in the middle for all clients for all congestion (except Vegas) possibly due to crowding at the server. Vegas tried to maintain the bandwidth consistency across all clients, by for example host 2 showed a dip in between 2-2.5s when both host 1 and 3 showed a rise. This kind of pattern continued till the end.



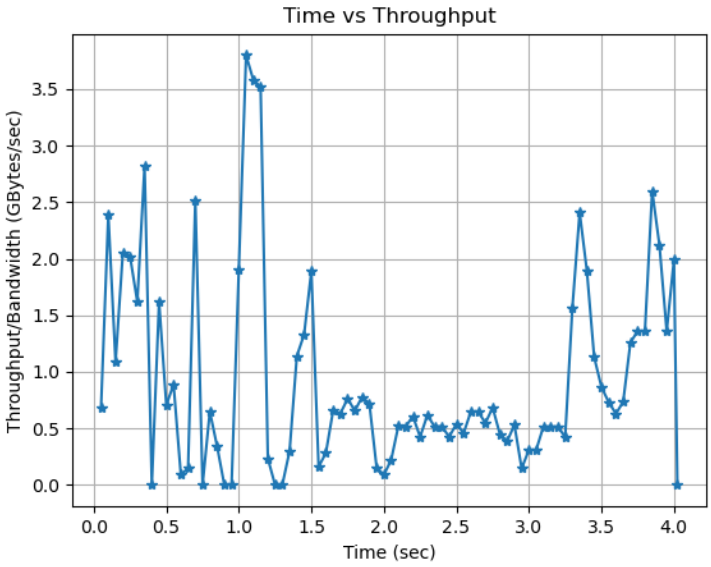
[config_c_host_1_congestion_BBR_loss_0.0]



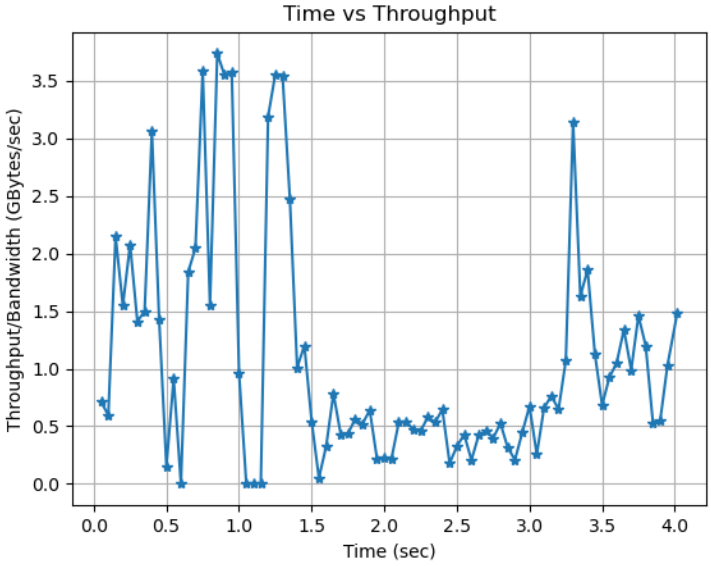
[config_c_host_2_congestion_BBR_loss_0.0]



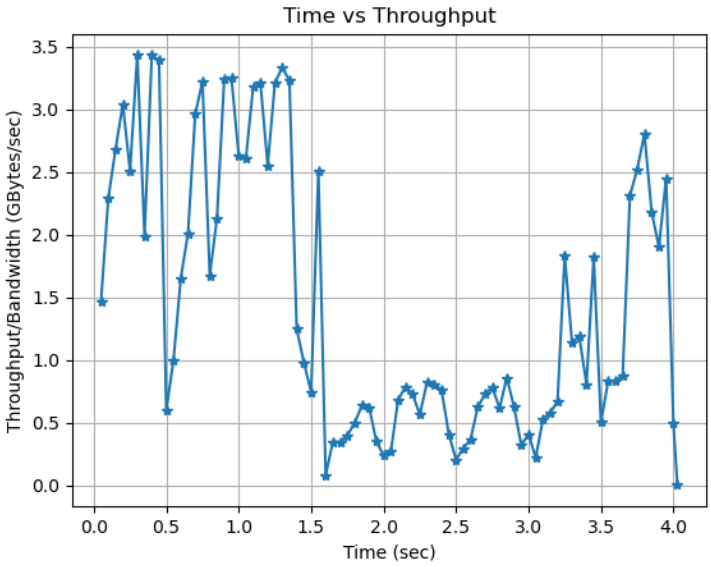
[config_c_host_3_congestion_BBR_loss_0.0]



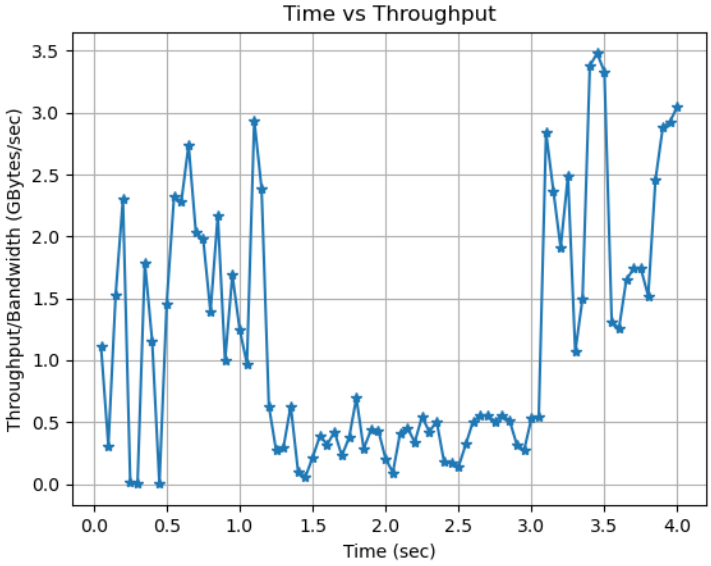
[config_c_host_1_congestion_Cubic_loss_0.0]



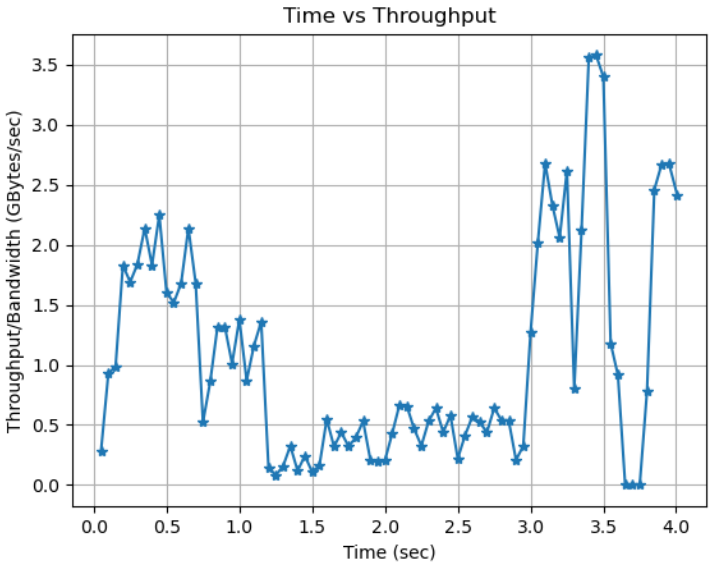
[config_c_host_2_congestion_Cubic_loss_0.0]



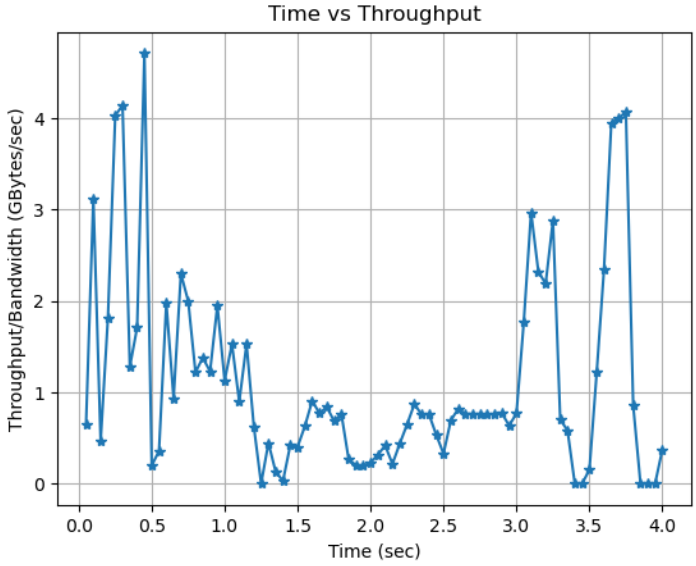
[config_c_host_3_congestion_Cubic_loss_0.0]



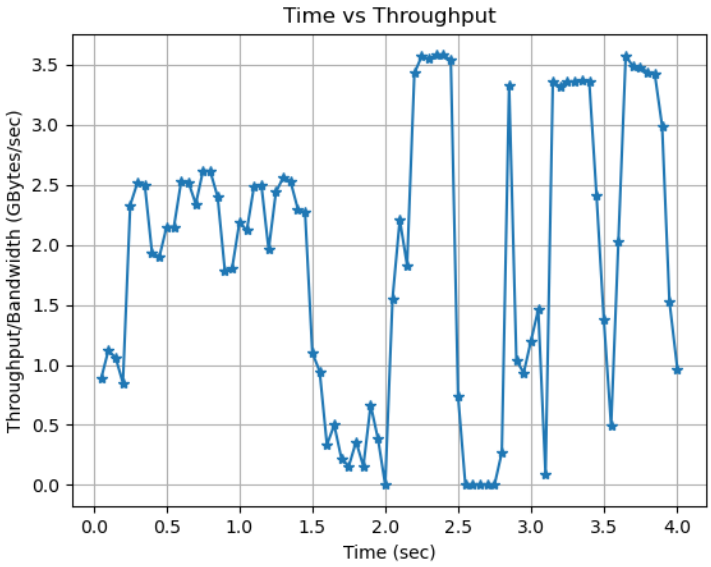
[config_c_host_1_congestion_Reno_loss_0.0]



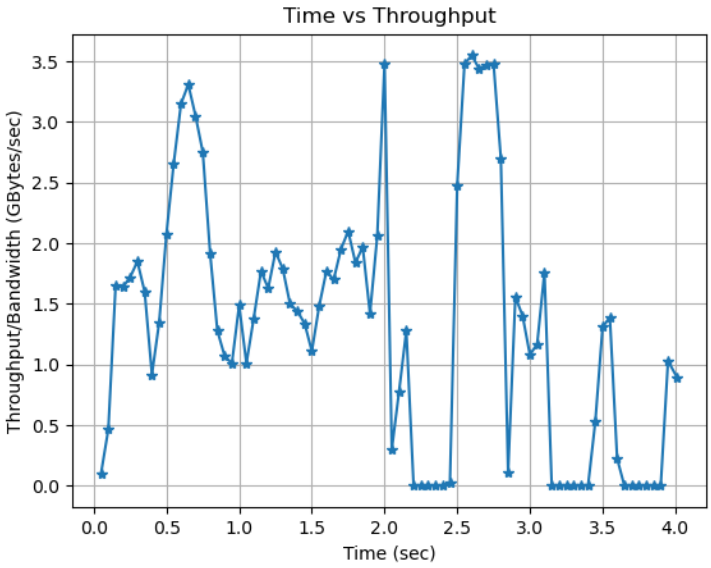
[config_c_host_2_congestion_Reno_loss_0.0]



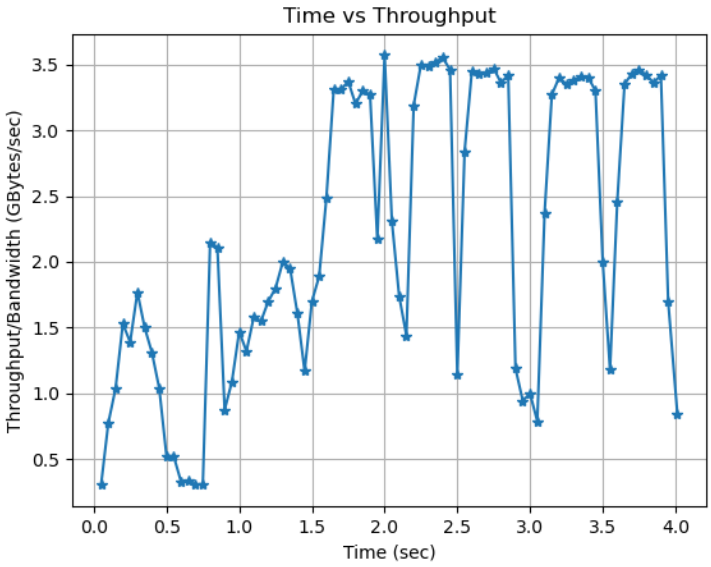
[config_c_host_3_congestion_Reno_loss_0.0]



[config_c_host_1_congestion_Vegas_loss_0.0]



[config_c_host_2_congestion_Vegas_loss_0.0]

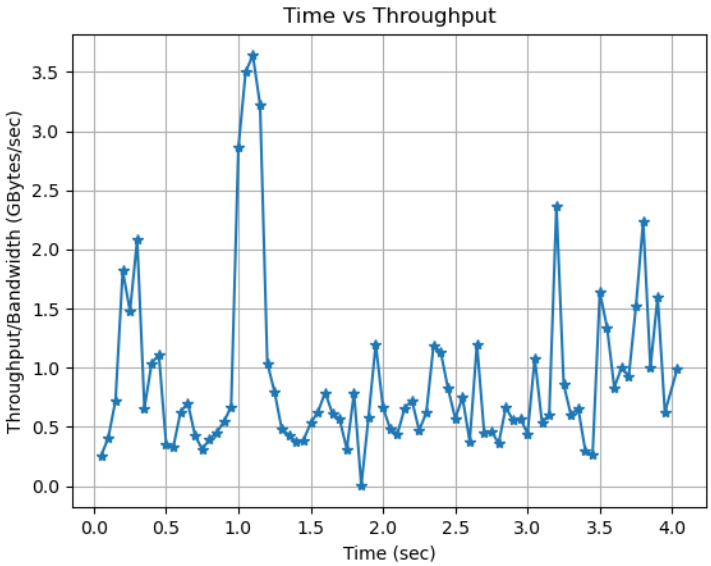


[config_c_host_3_congestion_Vegas_loss_0.0]

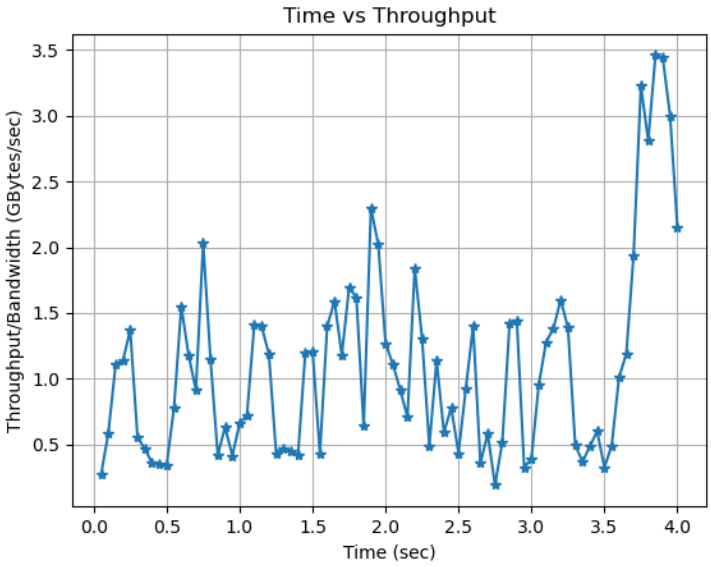
d. Configure the link loss parameter of the middle link (s1 - s2) to 1% and 3% and run the experiment in (b). Report and compare the throughput over time for each congestion control scheme. **(10 points)**

Answer:

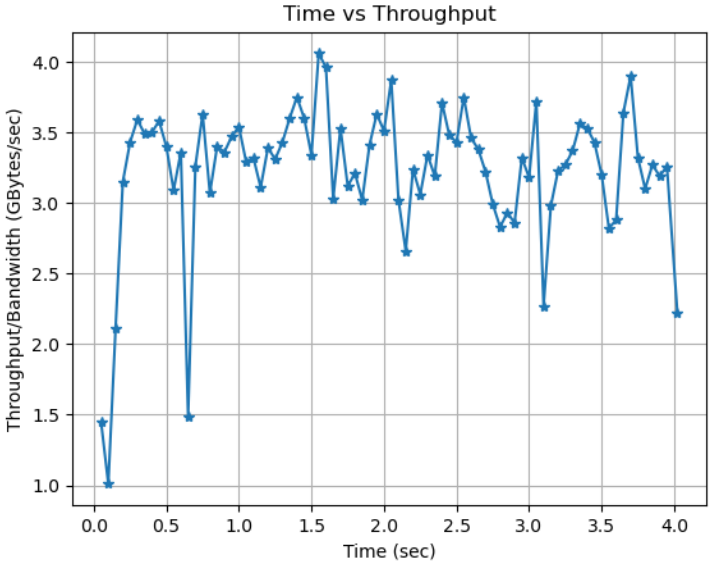
Here, we observed the 3% loss showed less throughput than 1% loss or at least there was an initial dip (in Vegas) or the initial peak was reduced in BBR.



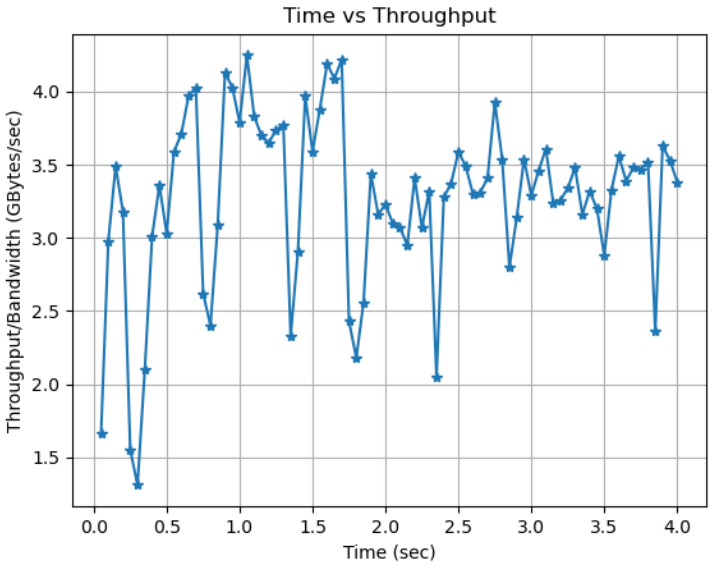
[config_b_host_1_congestion_BBR_loss_1.0]



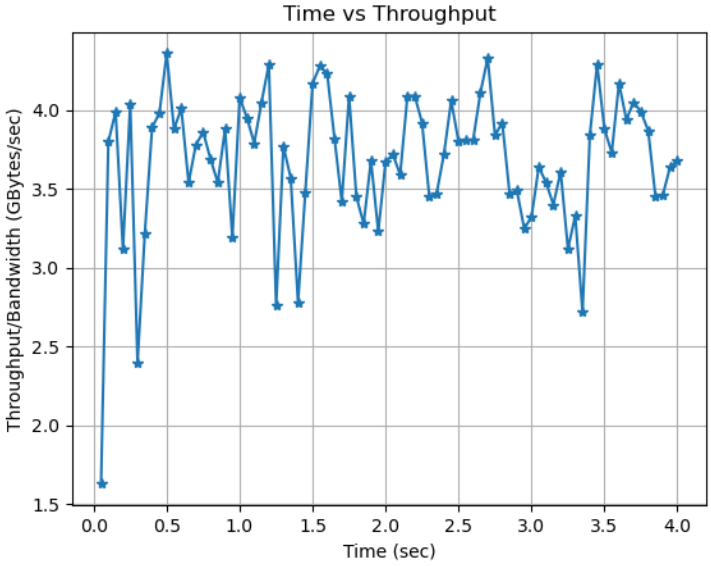
[config_b_host_1_congestion_BBR_loss_3.0]



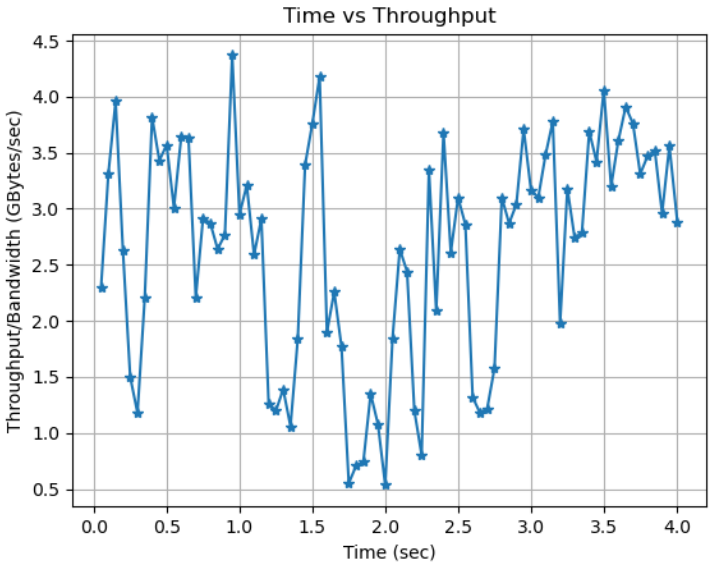
[config_b_host_1_congestion_Cubic_loss_1.0]



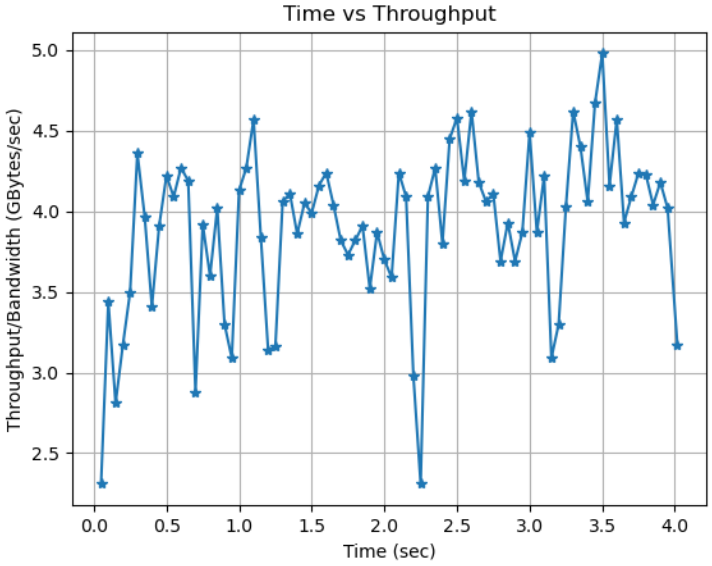
[config_b_host_1_congestion_Cubic_loss_3.0]



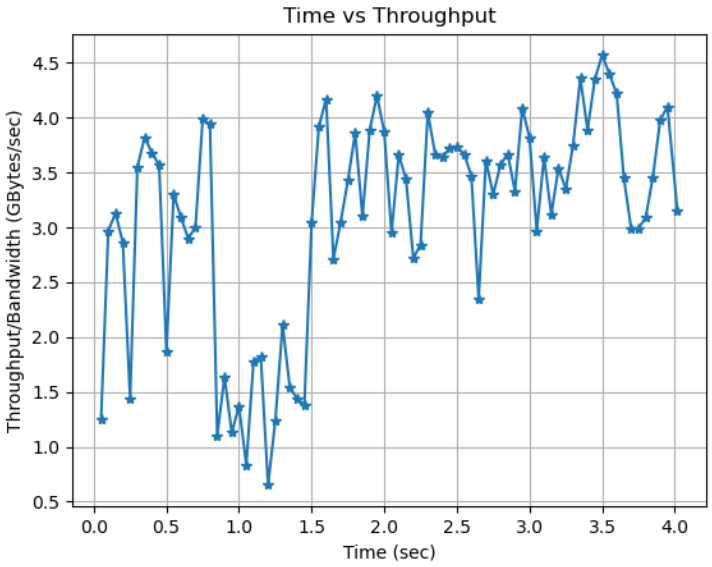
[config_b_host_1_congestion_Reno_loss_1.0]



[config_b_host_1_congestion_Reno_loss_3.0]



[config_b_host_1_congestion_Vegas_loss_1.0]



[config_b_host_1_congestion_Vegas_loss_3.0]