CS 433 Computer Networks (2023-24)

Assignment-2

Submitted: 11-Nov-2023

Name: Dhruv Gupta, Hirva Patel
Roll No.: 21110070, 21110154

Question 1:

(a) The implementation of the given topology has been done and it can seen in the given image. All the hosts and routers are able to ping all the other hosts and routers. The python code for this question is in the file '**ques1.py**'.

```
dhruv@dhruv-VirtualBox:~/Downloads$ sudo python test.py
*** Creating network
*** Adding controller
*** Adding hosts:
d1 d2 d3 d4 d5 d6 r1 r2 r3
*** Adding switches:
s1 s3 s5
*** Adding links:
(d1, s1) (d2, s1) (d3, s3) (d4, s3) (d5, s5) (d6, s5) (r1, r2) (r1, r3) (r2, r3) (s1, r1) (s3, r2) (
5, r3)
*** Configuring hosts
d1 d2 d3 d4 d5 d6 r1 r2 r3
*** Starting controller
c0
*** Starting 3 switches
s1 s3 s5 ...
*** Starting CLI:
mininet> pingall
*** Ping: testing ping reachability
d1 -> d2 d3 d4 d5 d6 r1 r2 r3
d2 -> d1 d3 d4 d5 d6 r1 r2 r3
d3 -> d1 d2 d4 d5 d6 r1 r2 r3
d4 -> d1 d2 d3 d5 d6 r1 r2 r3
d5 -> d1 d2 d3 d4 d6 r1 r2 r3
d6 -> d1 d2 d3 d4 d5 r1 r2 r3
r1 -> d1 d2 d3 d4 d5 d6 r2 r3
r2 -> d1 d2 d3 d4 d5 d6 r1 r3
r3 -> d1 d2 d3 d4 d5 d6 r1 r2
```

(b)

When d1 pings r1:

1	0.000000000	10.0.0.251	10.0.0.1	ICMP
2	0.000019039	10.0.0.1	10.0.0.251	ICMP
3	1.009398906	10.0.0.251	10.0.0.1	ICMP
4	1.009415286	10.0.0.1	10.0.0.251	ICMP
5	2.033288980	10.0.0.251	10.0.0.1	ICMP
6	2.033308352	10.0.0.1	10.0.0.251	ICMP

When d3 pings r1:

3	0.000025753	10.1.0.251	10.0.0.1	ICMP
4	0.000039191	10.0.0.1	10.1.0.251	ICMP
5	0.998142258	10.1.0.251	10.0.0.1	ICMP
6	0.998156263	10.0.0.1	10.1.0.251	ICMP
7	2.009699414	10.1.0.251	10.0.0.1	ICMP
8	2.009712067	10.0.0.1	10.1.0.251	ICMP

Here the packets come from the interface that is connected to router r2 (present in subnet of d3)

When d6 pings r1:

4	6.032303097	10.2.0.251	10.0.0.1	ICMP
5	6.033278718	10.0.0.1	10.2.0.251	ICMP
6	7.032336910	10.2.0.251	10.0.0.1	ICMP
7	7.032353088	10.0.0.1	10.2.0.251	ICMP
8	8.060233844	10.2.0.251	10.0.0.1	ICMP
9	8.060247588	10.0.0.1	10.2.0.251	ICMP

Here the packets come from the interface that is connected to router r3 (present in subnet of d5)

(c)

For the default route directly from r1 to r3:

```
info(net['r1'].cmd("ip route add 10.2.0.0/24 via 10.102.0.6 dev r1r3"))
#info(net['r1'].cmd("ip route add 10.2.0.0/24 via 10.100.0.2 dev r1r2"))
```

To change the route through r2 make the following changes in the code:

```
#info(net['r1'].cmd("ip route add 10.2.0.0/24 via 10.102.0.6 dev r1r3"))
info(net['r1'].cmd("ip route add 10.2.0.0/24 via 10.100.0.2 dev r1r2"))
```

When d1 pings d6 with the default route that is directly from r1 to r3:

```
mininet> d1 ping d6
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=2.40 ms
64 bytes from 10.2.0.252: icmp_seq=2 ttl=62 time=0.329 ms
64 bytes from 10.2.0.252: icmp_seq=3 ttl=62 time=0.084 ms
64 bytes from 10.2.0.252: icmp_seq=4 ttl=62 time=0.078 ms
```

```
mininet> d1 traceroute d6
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)  4.674 ms  6.937 ms  7.058 ms
 2  10.102.0.6 (10.102.0.6)  7.119 ms  7.292 ms  7.355 ms
 3  10.2.0.252 (10.2.0.252)  10.972 ms  11.100 ms  11.112 ms
```

When d1 pings d6 with the route through r2:

```
mininet> d1 ping d6
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=3.53 ms
64 bytes from 10.2.0.252: icmp_seq=2 ttl=62 time=1.05 ms
64 bytes from 10.2.0.252: icmp_seq=3 ttl=62 time=0.067 ms
64 bytes from 10.2.0.252: icmp_seq=4 ttl=62 time=0.168 ms
64 bytes from 10.2.0.252: icmp_seq=5 ttl=62 time=0.085 ms
```

```
mininet> d1 traceroute d6
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)  2.299 ms * *
 2  10.100.0.2 (10.100.0.2)  22.422 ms  22.783 ms  22.804 ms
 3  10.102.0.6 (10.102.0.6)  22.846 ms  23.364 ms  23.576 ms
 4  10.2.0.252 (10.2.0.252)  27.520 ms  27.760 ms  27.827 ms
```

It can be clearly seen that there is a latency difference between the 2 cases when the packets are transffered via r2 and when packets directly go from r1 to r3.

(d)

Routing tables for part a:

Route table of r1:

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	0.0.0.0	255.255.255.0	U	0	0	0	s1r1
10.1.0.0	10.100.0.2	255.255.255.0	UG	0	0	0	r1r2
10.2.0.0	10.102.0.6	255.255.255.0	UG	0	0	0	r1r3
10.100.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r1r2
10.102.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r1r3

Route table of r2:

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	10.100.0.1	255.255.255.0	UG	0	0	0	r2r1
10.1.0.0	0.0.0.0	255.255.255.0	U	0	0	0	s3r2
10.2.0.0	10.101.0.4	255.255.255.0	UG	0	0	0	r2r3
10.100.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r2r1
10.101.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r2r3

Route table of r3:

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	10.102.0.5	255.255.255.0	UG	0	0	0	r3r1
10.1.0.0	10.101.0.3	255.255.255.0	UG	0	0	0	r3r2
10.2.0.0	0.0.0.0	255.255.255.0	U	0	0	0	s5r3
10.101.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r3r2
10.102.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r3r1

Route tables for part (c) :

Route table of r1:

```
mininet> r1 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	s1r1
10.1.0.0	10.100.0.2	255.255.255.0	UG	0	0	0	r1r2
10.2.0.0	10.100.0.2	255.255.255.0	UG	0	0	0	r1r2
10.100.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r1r2
10.102.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r1r3

Route table of r2:

```
mininet> r2 route
Kernel IP routing table
```

Destination	Gateway	Genmask	Flags	Metric	Ref	Use	Iface
10.0.0.0	10.100.0.1	255.255.255.0	UG	0	0	0	r2r1
10.1.0.0	0.0.0.0	255.255.255.0	U	0	0	0	s3r2
10.2.0.0	10.101.0.4	255.255.255.0	UG	0	0	0	r2r3
10.100.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r2r1
10.101.0.0	0.0.0.0	255.255.255.0	U	0	0	0	r2r3

Rout table of r3:

```
mininet> r3 route
Kernel IP routing table
Destination     Gateway         Genmask         Flags Metric Ref    Use Iface
10.0.0.0        10.102.0.5      255.255.255.0   UG    0      0        0 r3r1
10.1.0.0        10.101.0.3      255.255.255.0   UG    0      0        0 r3r2
10.2.0.0        0.0.0.0         255.255.255.0   U     0      0        0 s5r3
10.101.0.0      0.0.0.0         255.255.255.0   U     0      0        0 r3r2
10.102.0.0      0.0.0.0         255.255.255.0   U     0      0        0 r3r1
```

Question 2:

The code to run in the terminal is named mini2.py. In case, it throws an error message “File already exists”, change the lines 29-30 from ‘s1-eth3’ and ‘s2-eth3’ to ‘s1-eth0’ and ‘s2-eth0’ or some other number that would work in your local machine. In case, all the eth(s) fail, restart your virtual machine and run again. This should solve the issue.

(a) The mininet topology is configured in mini2.py. Here is the output of net and pingall commands on mininet CLI.

```
┌──(parallels㉿kali-linux-2022-2)-[~]
└─$ sudo -E python mini2.py
[sudo] password for parallels:
mininet> pingall
*** Ping: testing ping reachability
h1 → h2 h3 h4
h2 → h1 h3 h4
h3 → h1 h2 h4
h4 → h1 h2 h3
*** Results: 0% dropped (12/12 received)
mininet> net
h1 h1-eth0:s1-eth1
h2 h2-eth0:s2-eth1
h3 h3-eth0:s1-eth2
h4 h4-eth0:s2-eth2
s1 lo:  s1-eth1:h1-eth0 s1-eth2:h3-eth0 s1-eth3:s2-eth3
s2 lo:  s2-eth1:h2-eth0 s2-eth2:h4-eth0 s2-eth3:s1-eth3
c0
```

Fig 2.1: Pingall and net cmnds

(b) In the mininet CLI, open xterms for h1 and h4, and run the iperf -s on h4 and iperf -c 10.0.0.4 (or the ip address assigned to h4 on your machine) -Z <type of tcp algorithm> -i 1 -t 20. Here, is a snapshot of the output on our machine.

```
┌──(root㉿kali-linux-2022-2)-[~]
└─# iperf -c 10.0.0.4 -Z vegas -t 20 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP congestion control set to vegas
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 35064 connected with 10.0.0.4 port 5001 (icwnd/mss/irt
t=14/1448/5455)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  7.13 GBytes  61.2 Gbits/sec
[  1] 1.0000-2.0000 sec  7.24 GBytes  62.2 Gbits/sec
[  1] 2.0000-3.0000 sec  7.29 GBytes  62.6 Gbits/sec
[  1] 3.0000-4.0000 sec  6.44 GBytes  55.3 Gbits/sec
[  1] 4.0000-5.0000 sec  2.54 GBytes  21.8 Gbits/sec
[  1] 5.0000-6.0000 sec  2.64 GBytes  22.7 Gbits/sec
[  1] 6.0000-7.0000 sec  5.96 GBytes  51.2 Gbits/sec
[  1] 7.0000-8.0000 sec  6.82 GBytes  58.6 Gbits/sec
[  1] 8.0000-9.0000 sec  6.65 GBytes  57.2 Gbits/sec
[  1] 9.0000-10.0000 sec  6.89 GBytes  59.2 Gbits/sec
[  1] 10.0000-11.0000 sec  6.20 GBytes  53.3 Gbits/sec
[  1] 11.0000-12.0000 sec  7.36 GBytes  63.3 Gbits/sec
[  1] 12.0000-13.0000 sec  7.35 GBytes  63.1 Gbits/sec
[  1] 13.0000-14.0000 sec  7.17 GBytes  61.6 Gbits/sec
[  1] 14.0000-15.0000 sec  7.03 GBytes  60.4 Gbits/sec
[  1] 15.0000-16.0000 sec  6.59 GBytes  56.6 Gbits/sec
[  1] 16.0000-17.0000 sec  6.16 GBytes  52.9 Gbits/sec
[  1] 17.0000-18.0000 sec  6.35 GBytes  54.6 Gbits/sec
[  1] 18.0000-19.0000 sec  7.05 GBytes  60.6 Gbits/sec
[  1] 19.0000-20.0000 sec  6.39 GBytes  54.9 Gbits/sec
[  1] 0.0000-20.0102 sec   127 GBytes  54.6 Gbits/sec

┌──(root㉿kali-linux-2022-2)-[~]
└─#
```

Fig 2.2: TCP vegas

```
┌──(root㉿kali-linux-2022-2)-[~]
└─# iperf -c 10.0.0.4 -Z reno -t 20 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP congestion control set to reno
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 50012 connected with 10.0.0.4 port 5001 (icwnd/mss/irtt=14/1448/2661)
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  7.21 GBytes  62.0 Gbits/sec
[  1] 1.0000-2.0000 sec  7.20 GBytes  61.8 Gbits/sec
[  1] 2.0000-3.0000 sec  6.31 GBytes  54.2 Gbits/sec
[  1] 3.0000-4.0000 sec  5.91 GBytes  50.7 Gbits/sec
[  1] 4.0000-5.0000 sec  3.70 GBytes  31.8 Gbits/sec
[  1] 5.0000-6.0000 sec  4.94 GBytes  42.4 Gbits/sec
[  1] 6.0000-7.0000 sec  5.15 GBytes  44.2 Gbits/sec
[  1] 7.0000-8.0000 sec  4.64 GBytes  39.9 Gbits/sec
[  1] 8.0000-9.0000 sec  6.61 GBytes  56.8 Gbits/sec
[  1] 9.0000-10.0000 sec  5.57 GBytes  47.9 Gbits/sec
[  1] 10.0000-11.0000 sec  7.14 GBytes  61.3 Gbits/sec
[  1] 11.0000-12.0000 sec  5.57 GBytes  47.8 Gbits/sec
[  1] 12.0000-13.0000 sec  5.30 GBytes  45.5 Gbits/sec
[  1] 13.0000-14.0000 sec  6.27 GBytes  53.9 Gbits/sec
[  1] 14.0000-15.0000 sec  7.20 GBytes  61.9 Gbits/sec
[  1] 15.0000-16.0000 sec  6.90 GBytes  59.2 Gbits/sec
[  1] 16.0000-17.0000 sec  6.48 GBytes  55.7 Gbits/sec
[  1] 17.0000-18.0000 sec  7.16 GBytes  61.5 Gbits/sec
[  1] 18.0000-19.0000 sec  7.35 GBytes  63.1 Gbits/sec
[  1] 19.0000-20.0000 sec  7.39 GBytes  63.5 Gbits/sec
[  1] 0.0000-20.0119 sec   124 GBytes  53.2 Gbits/sec
```

Fig 2.3: TCP reno

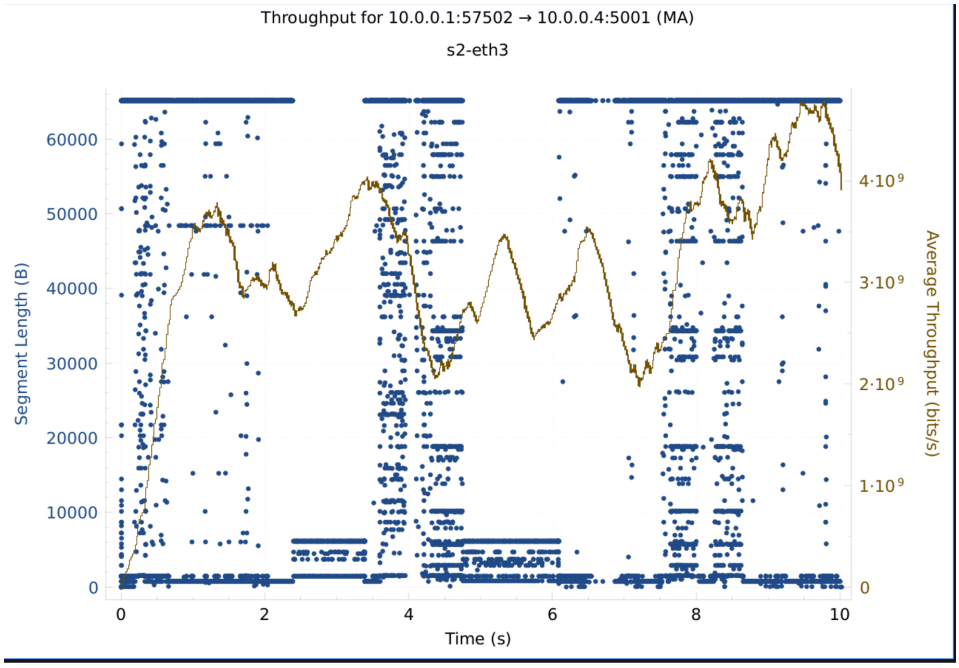
```
┌──(root㉿kali-linux-2022-2)-[~]
└─# iperf -c 10.0.0.4 -Z cubic -t 20 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP congestion control set to cubic
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 46208 connected with 10.0.0.4 port 5001 (icwnd/ms
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  7.13 GBytes  61.3 Gbits/sec
[  1] 1.0000-2.0000 sec  7.21 GBytes  61.9 Gbits/sec
[  1] 2.0000-3.0000 sec  7.20 GBytes  61.8 Gbits/sec
[  1] 3.0000-4.0000 sec  7.25 GBytes  62.2 Gbits/sec
[  1] 4.0000-5.0000 sec  7.30 GBytes  62.7 Gbits/sec
[  1] 5.0000-6.0000 sec  6.78 GBytes  58.2 Gbits/sec
[  1] 6.0000-7.0000 sec  7.19 GBytes  61.8 Gbits/sec
[  1] 7.0000-8.0000 sec  7.32 GBytes  62.9 Gbits/sec
[  1] 8.0000-9.0000 sec  7.30 GBytes  62.7 Gbits/sec
[  1] 9.0000-10.0000 sec  7.25 GBytes  62.3 Gbits/sec
[  1] 10.0000-11.0000 sec  7.24 GBytes  62.2 Gbits/sec
[  1] 11.0000-12.0000 sec  7.18 GBytes  61.7 Gbits/sec
[  1] 12.0000-13.0000 sec  6.65 GBytes  57.1 Gbits/sec
[  1] 13.0000-14.0000 sec  7.12 GBytes  61.2 Gbits/sec
[  1] 14.0000-15.0000 sec  7.22 GBytes  62.0 Gbits/sec
[  1] 15.0000-16.0000 sec  7.14 GBytes  61.3 Gbits/sec
[  1] 16.0000-17.0000 sec  7.13 GBytes  61.3 Gbits/sec
[  1] 17.0000-18.0000 sec  7.06 GBytes  60.7 Gbits/sec
[  1] 18.0000-19.0000 sec  7.25 GBytes  62.3 Gbits/sec
[  1] 19.0000-20.0000 sec  7.24 GBytes  62.2 Gbits/sec
[  1] 0.0000-20.0120 sec   143 GBytes  61.4 Gbits/sec
```

Fig 2.4: TCP cubic

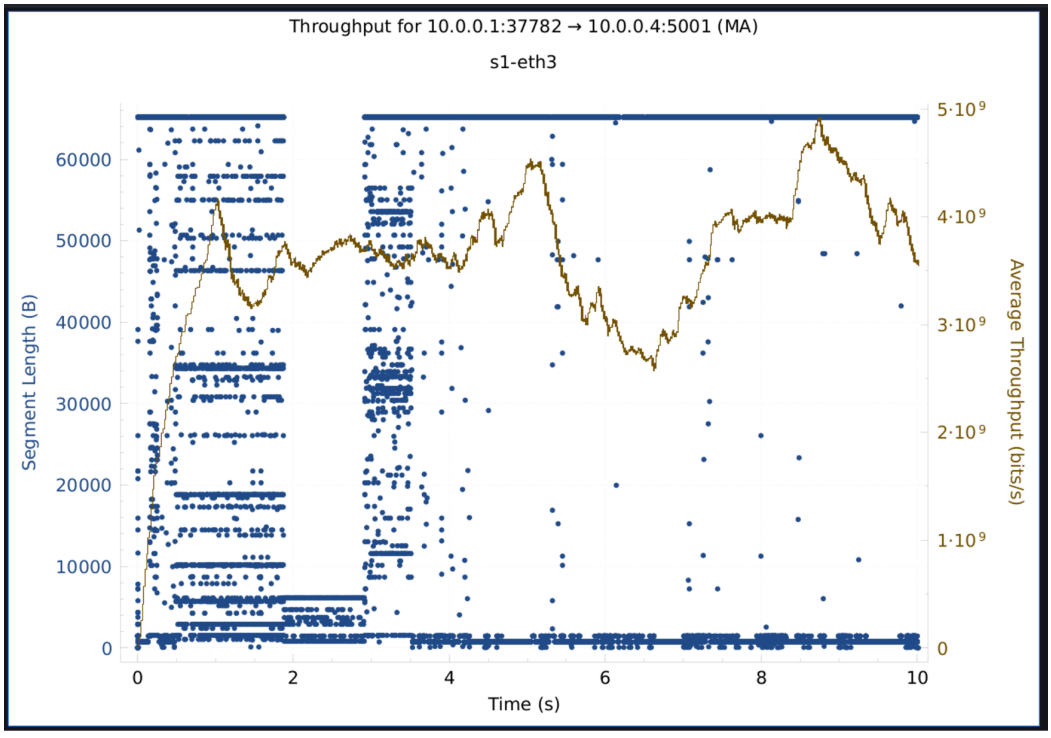
```
┌──(root㉿kali-linux-2022-2)-[~]
└─# iperf -c 10.0.0.4 -Z bbr -t 20 -i 1
------------------------------------------------------------
Client connecting to 10.0.0.4, TCP port 5001
TCP congestion control set to bbr
TCP window size: 85.3 KByte (default)
------------------------------------------------------------
[  1] local 10.0.0.1 port 57518 connected with 10.0.0.4 port 5001 (icwnd/ms
[ ID] Interval       Transfer     Bandwidth
[  1] 0.0000-1.0000 sec  4.52 GBytes  38.8 Gbits/sec
[  1] 1.0000-2.0000 sec  4.39 GBytes  37.7 Gbits/sec
[  1] 2.0000-3.0000 sec  4.60 GBytes  39.5 Gbits/sec
[  1] 3.0000-4.0000 sec  4.54 GBytes  39.0 Gbits/sec
[  1] 4.0000-5.0000 sec  4.50 GBytes  38.7 Gbits/sec
[  1] 5.0000-6.0000 sec  4.59 GBytes  39.4 Gbits/sec
[  1] 6.0000-7.0000 sec  4.57 GBytes  39.2 Gbits/sec
[  1] 7.0000-8.0000 sec  4.58 GBytes  39.3 Gbits/sec
[  1] 8.0000-9.0000 sec  4.51 GBytes  38.7 Gbits/sec
[  1] 9.0000-10.0000 sec  4.58 GBytes  39.4 Gbits/sec
[  1] 10.0000-11.0000 sec  4.59 GBytes  39.4 Gbits/sec
[  1] 11.0000-12.0000 sec  4.57 GBytes  39.3 Gbits/sec
[  1] 12.0000-13.0000 sec  4.60 GBytes  39.5 Gbits/sec
[  1] 13.0000-14.0000 sec  4.60 GBytes  39.5 Gbits/sec
[  1] 14.0000-15.0000 sec  4.58 GBytes  39.3 Gbits/sec
[  1] 15.0000-16.0000 sec  4.59 GBytes  39.5 Gbits/sec
[  1] 16.0000-17.0000 sec  4.55 GBytes  39.1 Gbits/sec
[  1] 17.0000-18.0000 sec  3.72 GBytes  31.9 Gbits/sec
[  1] 18.0000-19.0000 sec  4.58 GBytes  39.3 Gbits/sec
[  1] 19.0000-20.0000 sec  4.59 GBytes  39.5 Gbits/sec
[  1] 0.0000-20.0089 sec  90.3 GBytes  38.8 Gbits/sec
```

Fig 2.5: TCP bbr

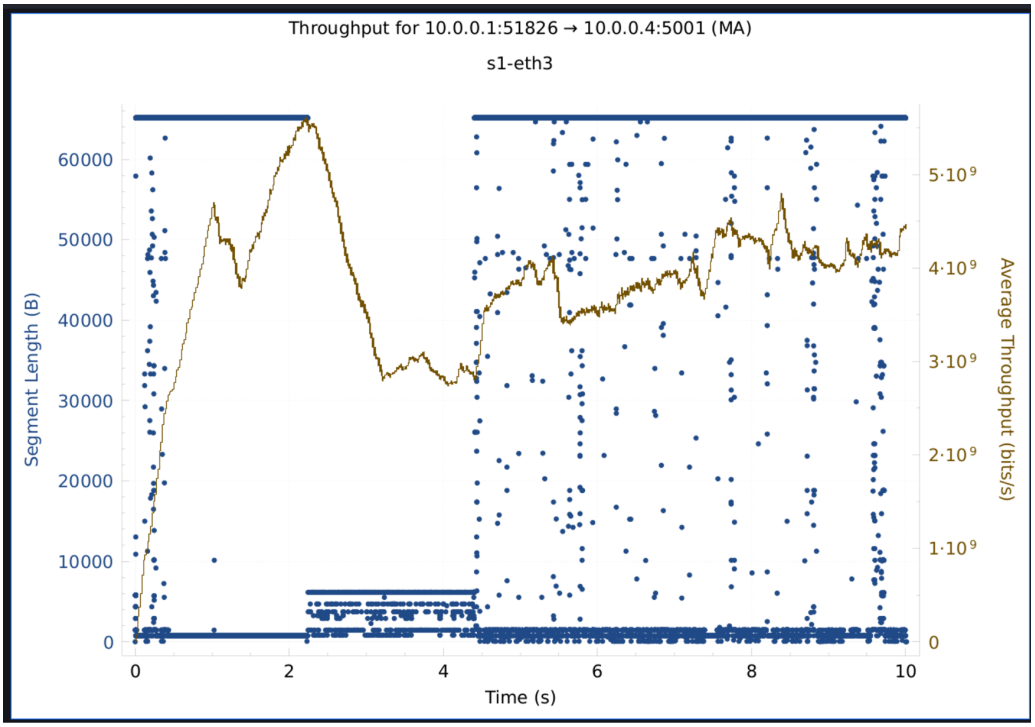
We can see in TCP vegas that it reduces its sending rate more aggressively. Thus, the throughput rate decreases to near 20Gbits/sec. In contrast, in TCP reno, the sending rate is cut in half when it faces congestion. We don't see any congestion even in TCP cubic data collected. However, it increases more aggressively after facing a congestion event. In TCP BBR, we see relatively lower throughput among all the algorithms. This can happen if the network conditions don't align with BBR's strengths or some inconsistencies in RTT measurement.



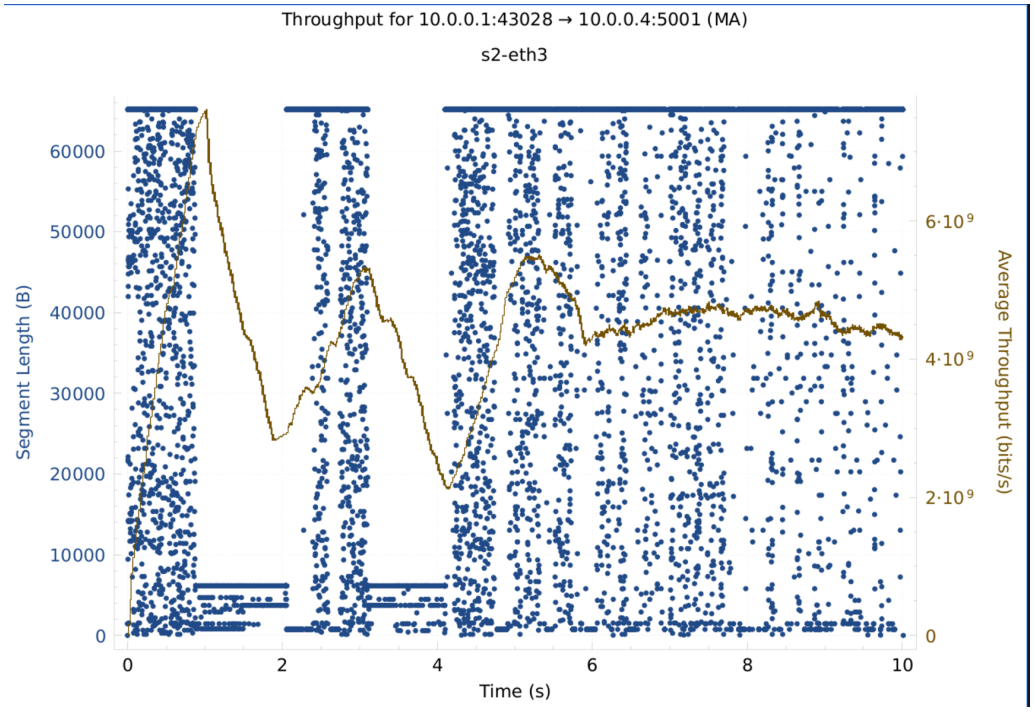
TCP vegas



TCP reno



TCP cubic



TCP BBR

(c) In the mininet CLI, open xterms for h1, h2, h3 and h4 and run the iperf -s command on xterm h4, and iperf -c 10.0.0.4 -Z <typeofalgorithm> -i 1 -t 20 from each xterms of h1, h2 and h3.

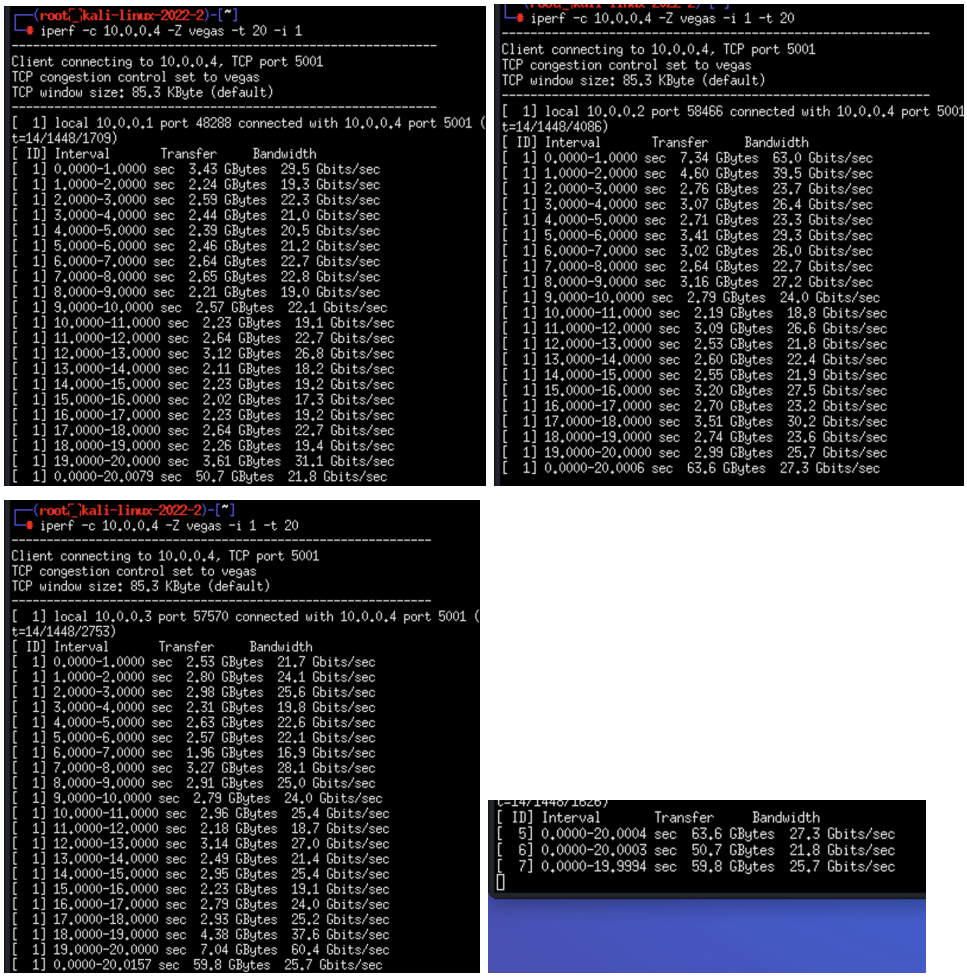


Fig 5: TCP vegas for h1, h2 and h3.

TCP Vegas uses a delay-based approach to achieve fairness. It aims to maintain a constant queueing delay by adjusting the congestion window based on the observed round-trip time (RTT). Connections with longer RTTs are allowed a larger congestion window, ensuring that they get a proportional share of bandwidth. Shorter RTTs result in smaller congestion windows. Thus, we can observe fair throughput sharing.

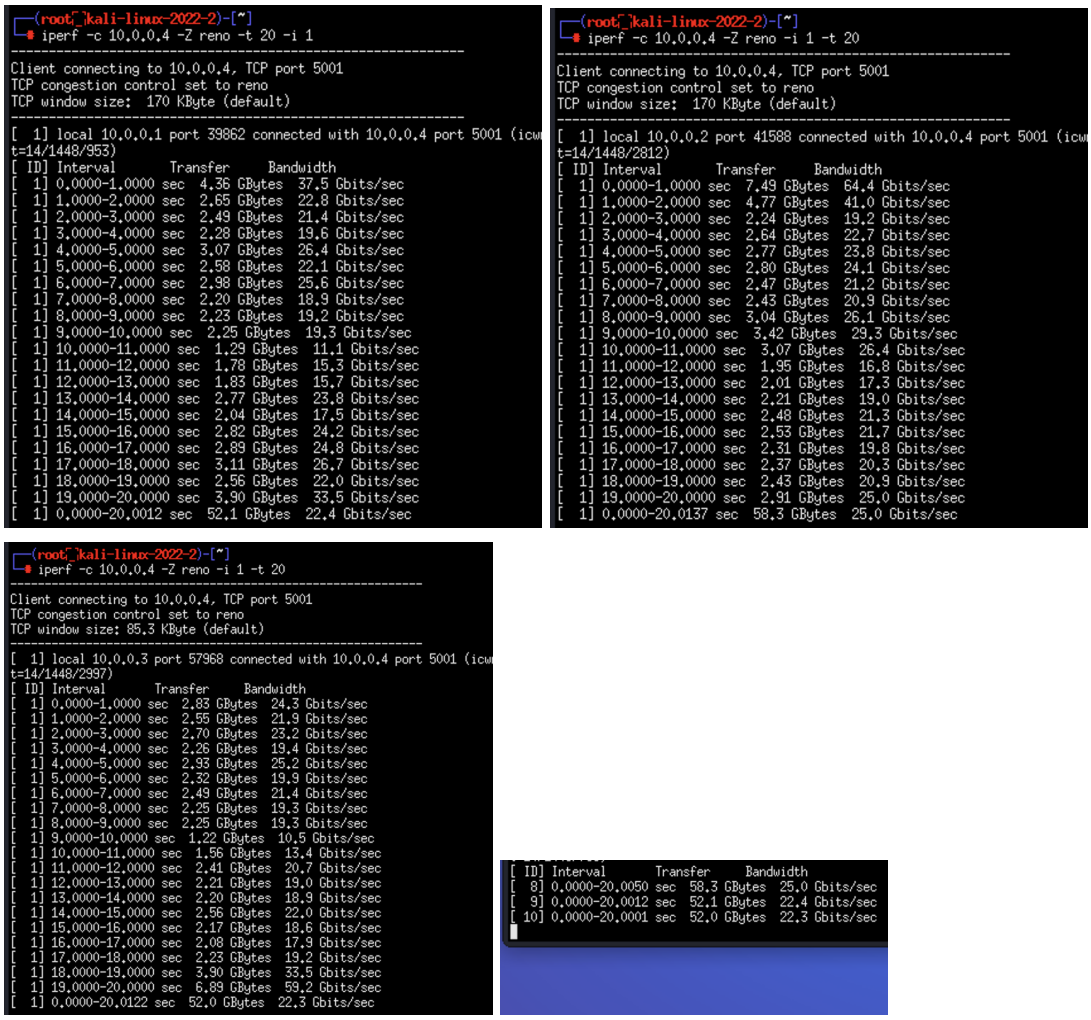


Fig 6: TCP reno: h1, h2 and h3 and throughputs of each.

TCP Reno achieves fairness through a combination of congestion window adjustment and additive increase/multiplicative decrease (AIMD) behavior. During congestion avoidance, each connection increases its congestion window additively, gaining a small share of bandwidth. When congestion is detected (packet loss), connections reduce their congestion window multiplicatively, leading to fair bandwidth sharing. Same as above, we see the fairness.

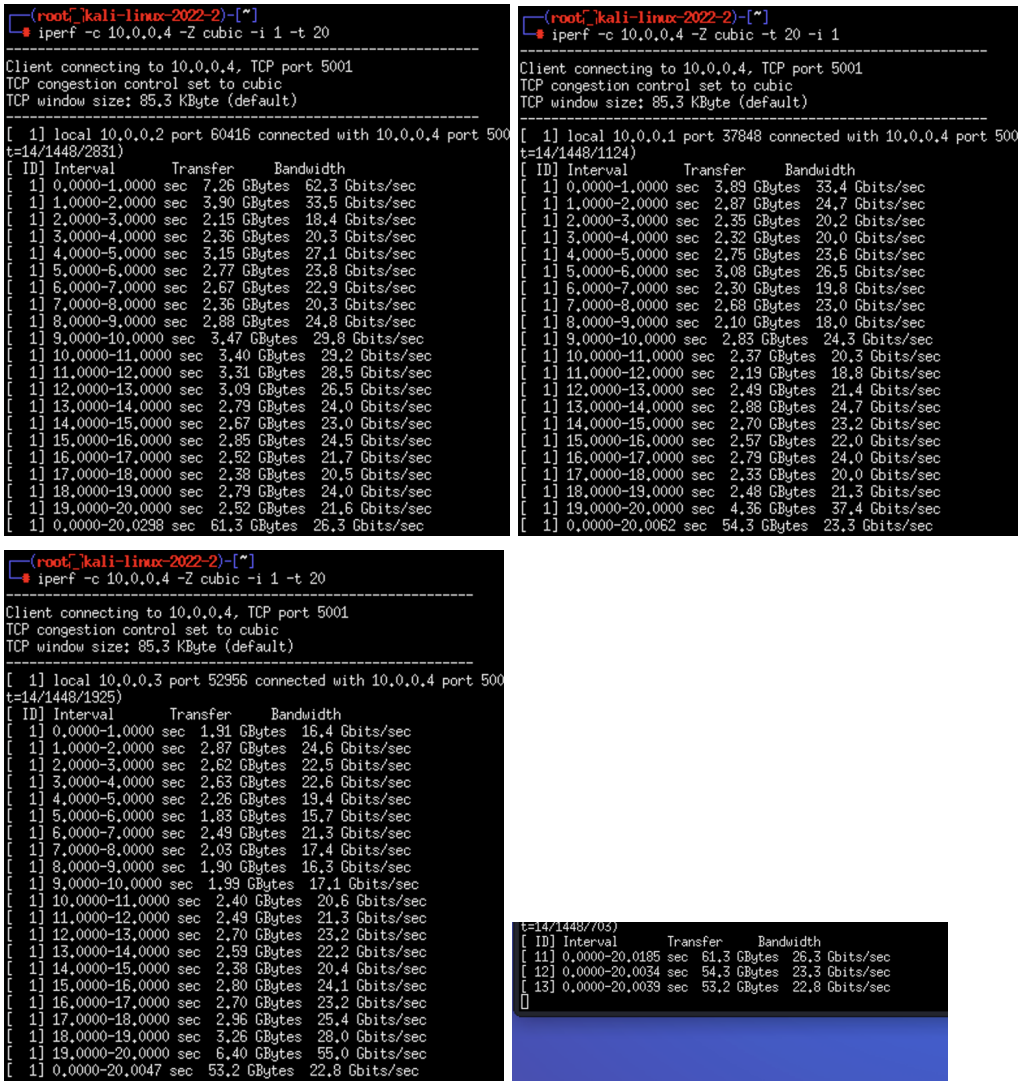


Fig 7: TCP cubic for h1, h2 and h3 and throughput for each.

TCP Cubic, like Reno, uses AIMD behavior, but it has a cubic-shaped increase function. Cubic is designed to achieve fairness by adjusting the sending rate more aggressively after congestion events, allowing it to recover more quickly and maintain a fair share of bandwidth.

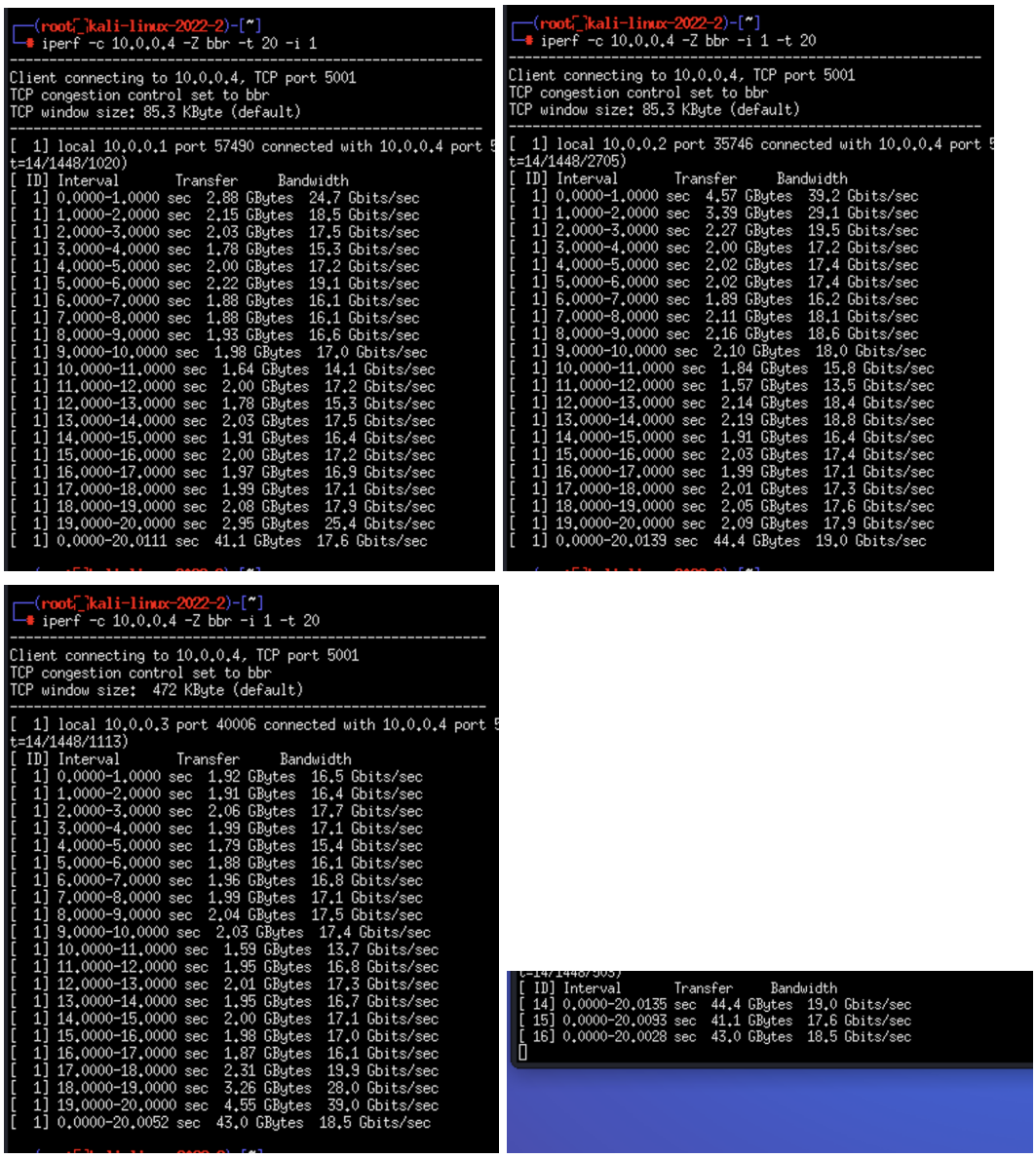


Fig 8: TCP bbr for h1, h2 and h3.

BBR uses a combination of bandwidth probing and RTT estimation to achieve fairness. BBR aims to divide throughput fairly by estimating the available bandwidth and adjusting its sending rate accordingly. The BBR algorithm adapts to the dynamics of the network, exploring available bandwidth during probing phases and maintaining a steady sending rate during congestion avoidance.

(d) Run the mini2d1.py for loss = 1% to find the throughput when the link loss percentage of the middle link is 1%. Similarly, run mini2d3.py for loss = 3%. Here, are the compared results.

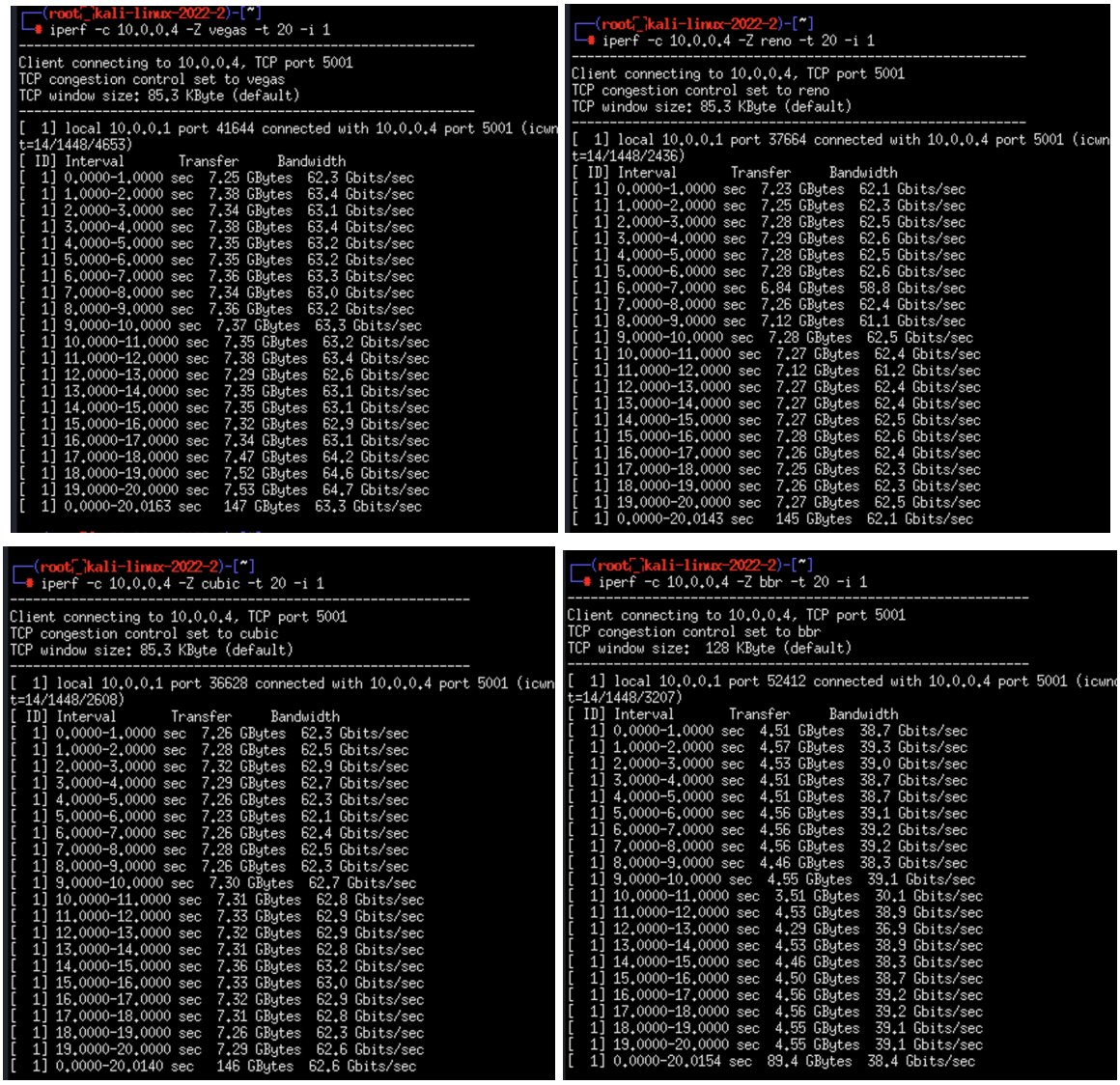


Fig 9: For loss=1%

With a link loss of 1%, congestion control algorithms, such as Cubic, Reno, Vegas, or BBR, are designed to handle occasional packet loss. The impact on throughput may be noticeable, but the algorithms are generally designed to recover from moderate levels of loss without a significant reduction in throughput.

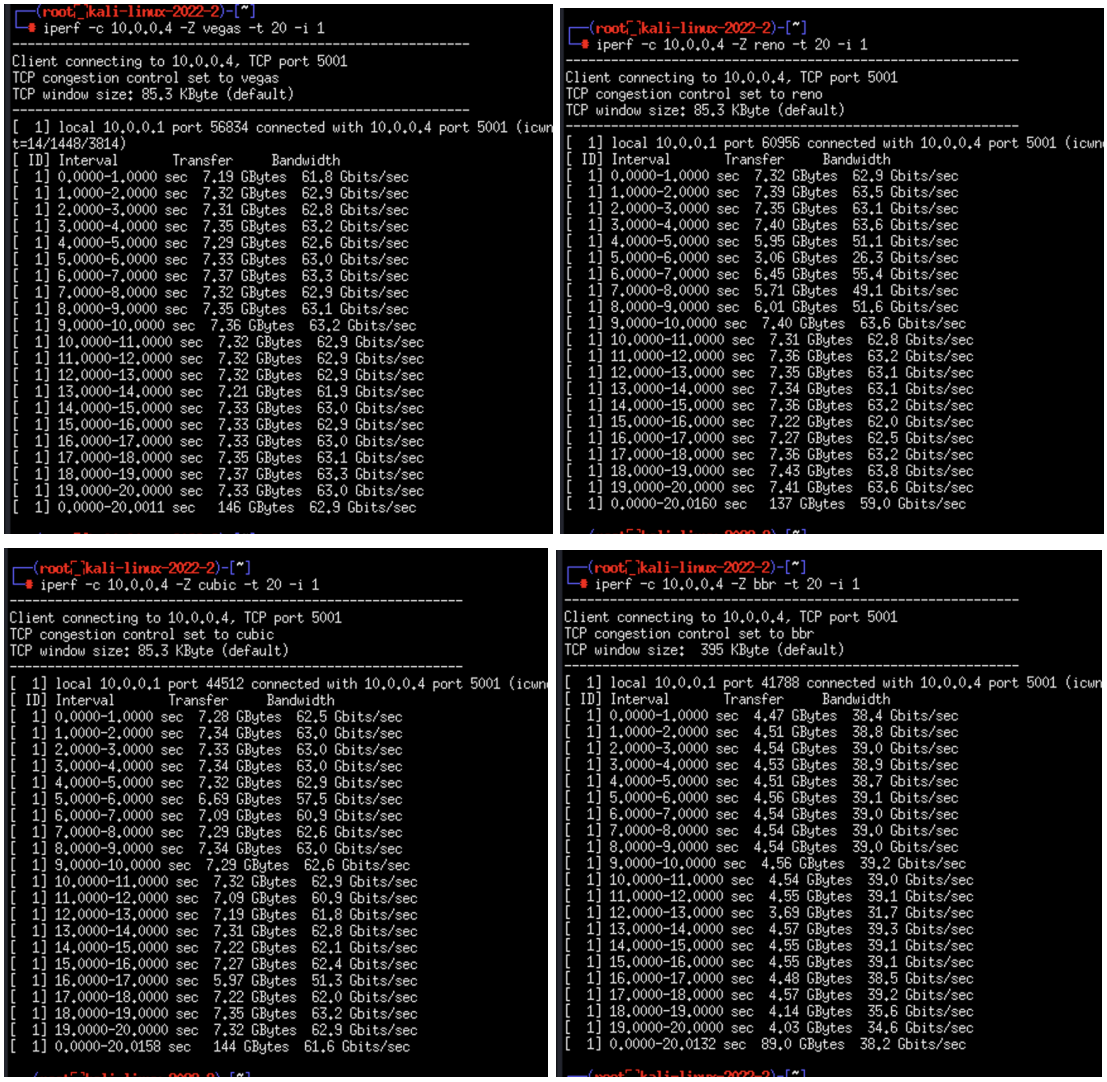


Fig 10: for loss 3%

A higher link loss of 3% is considered more significant, and congestion control algorithms are likely to respond more aggressively to mitigate congestion. The algorithms may reduce their sending rate more substantially, leading to a potential decrease in throughput.

TCP Reno and Cubic may exhibit a more aggressive response to packet loss. TCP Vegas, being delay-based, may have a different response compared to traditional loss-based algorithms. TCP BBR is designed to recover quickly from losses and maintain high throughput, but the specific response can depend on the version and tuning.

The overall impact on throughput is influenced by other network conditions, such as available bandwidth, round-trip time (RTT), and the presence of competing flows.