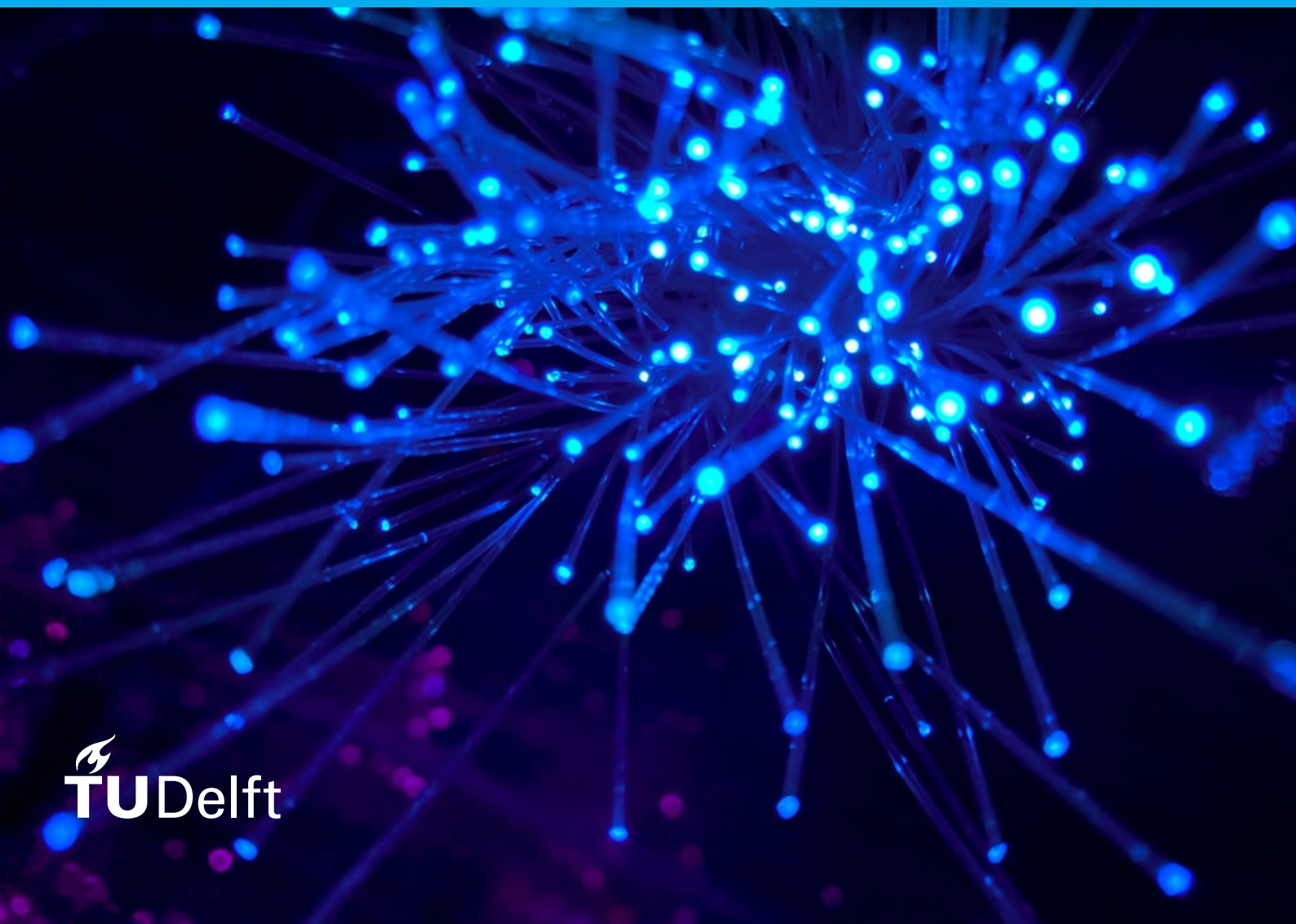
CS4055
High Performance Data Networking
Project Report

Comparative Study of P4 and OpenFlow monitoring application

Group 2



TU Delft

CS4055 High Performance Data Networking Project Report

Comparative Study of P4 and OpenFlow monitoring application

March 22, 2021

TU Delft

Contents

Achievements

OpenFlow

- Extended ***Monitoring_Routing_Bandwidth_Delay.py***[1] with an accurate delay monitoring function and a packet loss monitoring function.
- Bandwidth Monitoring: Tested and analyzed the bandwidth monitoring function with different network configurations including different bandwidth settings, different topology and multiple simultaneous flows.
- Delay Monitoring: Calculated the delay by adding additional flow request message to remove the round trip time between the OpenFlow controller and switches. Tested and analyzed the delay monitoring function with different network configurations including different delay settings, different topology.
- Loss Monitoring: Calculated the packet loss by the returned statistics from flow requests messages and probe message. Tested and analyzed the loss monitoring function with different network configurations including different loss settings, different topology and different protocols.

P4

- Completed the basic P4 network forwarding functionality
- Delay Monitoring: show which switch(es) in the network is slowest and is the bottle neck in a network path
- Bandwidth Monitoring: monitor amount of packets on a path and reduce overloaded traffics to maintain bandwidth balance in a network
- Loss Monitoring: show at which switches most packets are dropped in the network path
- P4 monitoring fully with SimpleSwitchGrpc and gRPC controller with P4Runtime support instead of SimpleSwitch
- Using experimental Thrift server feature to enable BMv2 debugger and runtime_CLI

Contribution

- OpenFlow Implementation (Hang)
- OpenFlow Testing (Hang & Shubhra)
- P4 Implementation proposal (Shubhra)

[1] This is the code example provided by the 2020-2021 Reader Solution for bandwidth monitoring and delay monitoring of the OpenFlow switch

- P4 Implementation (Tat)
- P4 Testing (Tat & Shubhra)
- Results Analysis (Tat, Hang, Shubhra)
- Comparison of P4 and OpenFlow (Shubhra)

1

Abstract

A network monitoring application collects network statistics such as propagation delay between source node and destination node, number of packets lost while traversing the route and bandwidth utilisation of the network. These applications help network designers in optimising network capacity, provide security analysis as well as give insights about the traffic which passes over the network[3]. A network monitoring application has been developed for both OpenFlow as well as P4 which effectively collects network data. This data will be used for comparing OpenFlow and P4 on the basis of the above stated network characteristics. This comparison will benefit the operators, who might be interested in implementing OpenFlow and P4.

2

OpenFlow Monitoring Implementation & Testing

2.1. OpenFlow Monitoring

In earlier networks, there was no separation of control plane and data plane which made those networks inefficient as adding new rules and updates was a challenge. With the emergence of Software Defined Networks(SDN), the data plane and control plane can be maintained virtually and function independent of each other[2]. OpenFlow is a communication protocol which implements this logical separation of data plane and control plane. In OpenFlow, switches are dumb forwarding devices and controller is responsible for installing all the rules. Using OpenFlow switch protocols these rules can be added, deleted and updated.

The OpenFlow Monitoring application is implemented on the basis of the monitoring application provided by the project reader of ***2020-2021 CS4055 High Performance Data Networking***. The code of our application extends ***Monitoring_Routing_Bandwidth_Delay.py***[1] with an accurate delay monitoring function and a packet loss function.

The OpenFlow Monitoring application is developed with the instant virtual network **Mininet** and the OpenFlow controller framework **Ryu**. The application is able to monitor the bandwidth, delay, and packet loss for data flows in a network. The system architecture of our application is shown in Figure 2.13. The control plane is able to route the data flow through network devices and collect QoS statistics. The variables of the monitoring application is declared in Table 2.1

[1] This is the code example provided by the 2020-2021 Reader Solution for bandwidth monitoring and delay monitoring of the OpenFlow switch

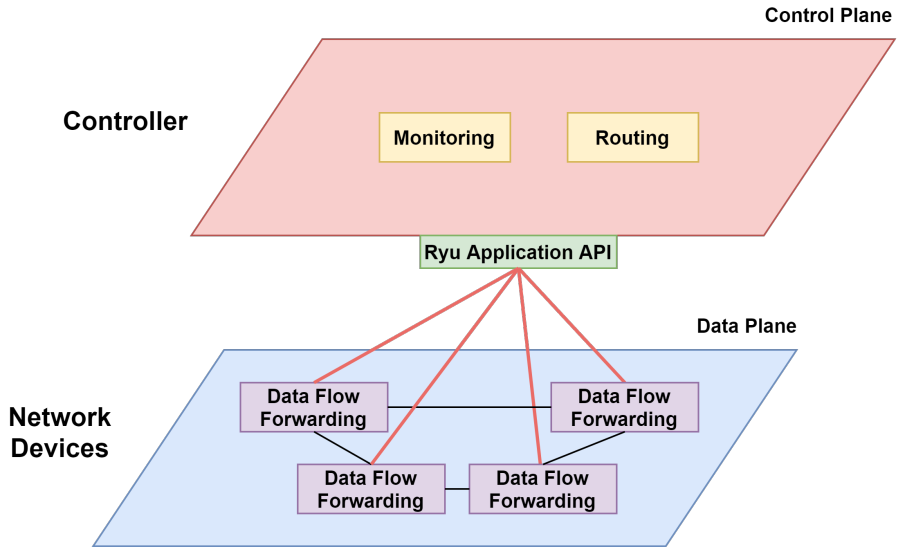


Figure 2.1: System Architecture

2.2. Bandwidth Monitoring

2.2.1. Implementation

The bandwidth of a data flow can be obtained by sending the **OFPFlowStatsRequest** message to the destination switch in the data path of the monitored data flow periodically (every **t_int**). It is presented in Figure 2.2.

variable	description
destinationSwitch	the destination switch in the data flow
sourceSwitch	the source switch in the data flow
src_mac	the mac address of the source switch in the data flow
dst_mac	the mac address of the destination switch in the data flow
sourcePort	the port from the first switch which the probe packet will be sent out to
delay	delay of the probe packet
throughput	the through put of the link between the source switch and the destination switch
bytes_last	the total bytes of the data flow at the previous timestamp
rtt_s_c_src	the round trip time between the source switch and the controller
rtt_s_c_dst	the round trip time between the destination switch and the controller
packet_src_last	the total packets sent by the source switch at the previous timestamp
packet_dst_last	the total packets received by the destination switch at the previous timestamp
packet_src	the total packets sent by the source switch during the polling cycle
packet_dst	the total packets received by the destination switch during the polling cycle
packet_loss	the packet loss during the polling cycle
src_timestamp	the current timestamp for the request message of the source switch
dst_timestamp	the current timestamp for the request message of the destination switch

Table 2.1: variables declaration for OpenFlow Monitoring Application

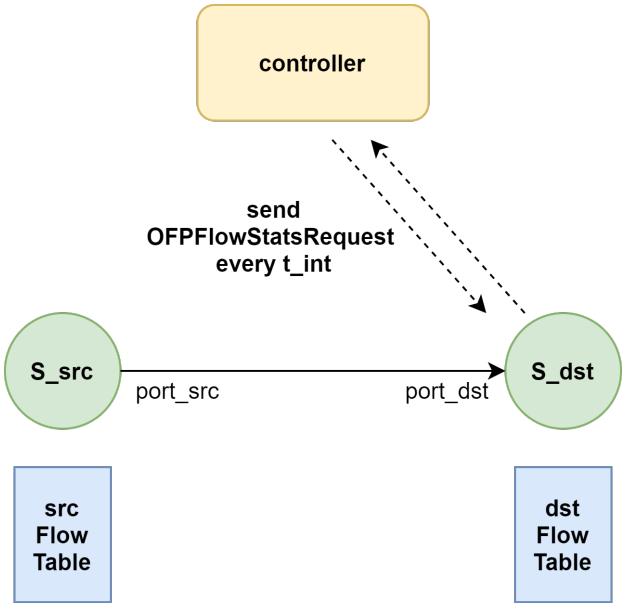


Figure 2.2: Bandwidth Monitoring

2.2.2. Testing

To verify the functionality of bandwidth monitoring, we test the application with different bandwidth settings with different topology as well as varying client host configurations. Linear topology and ring topology are used in tests. Examples of the two topology are shown in Figure 2.3 and Figure 2.4. The example command for those tests is shown below:

```
sudo mn --mac --arp --topo=linear,5 --controller=remote \
--switch=ovs,protocol=OpenFlow13 \
--links=tc,bw=20
```

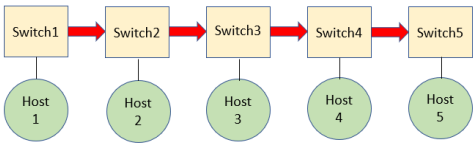


Figure 2.3: Linear Topology

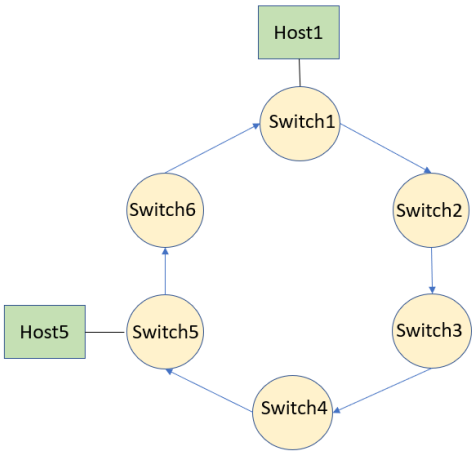


Figure 2.4: Ring Topology

Table 2.2 gives an overview of all the tests for bandwidth monitoring. In Figure 2.7 and Figure 2.11, the client of server h1 is h2, the client of server h4 is h5.

Figure	Topology	Bandwidth (Mbps)	Server	Client
2.5	Linear	20	h1	h2
2.6	Linear	20	h1	h5
2.7	Linear	20	h1, h4	h2, h5
2.8	Ring	20	h1	h2
2.9	Linear	40	h1	h2
2.10	Linear	40	h1	h5
2.11	Linear	40	h1, h4	h2, h5
2.12	Ring	40	h1	h2

Table 2.2: Bandwidth Monitoring Tests

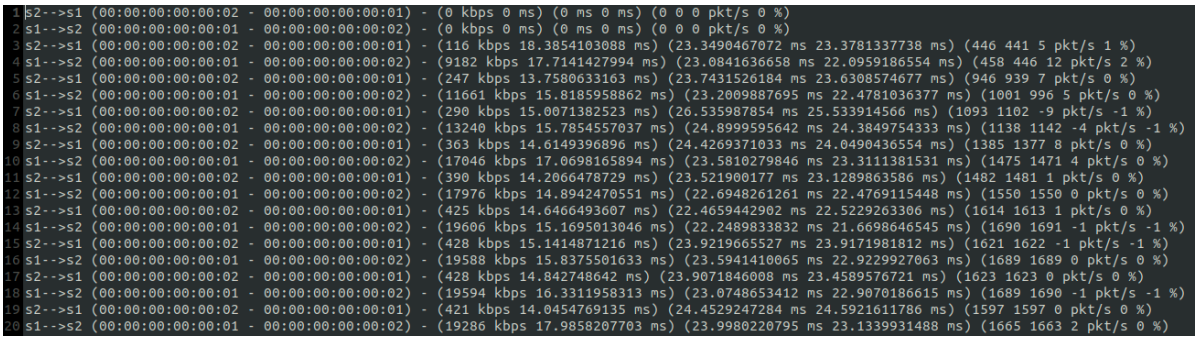


Figure 2.5: Bandwidth 20 Mbps, Linear Topology, server h1, client h2

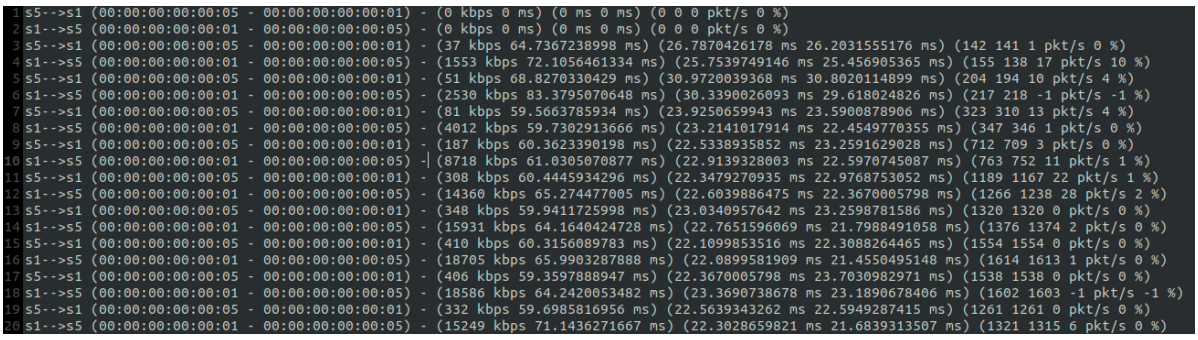


Figure 2.6: Bandwidth 20 Mbps, Linear Topology, server h1, client h5

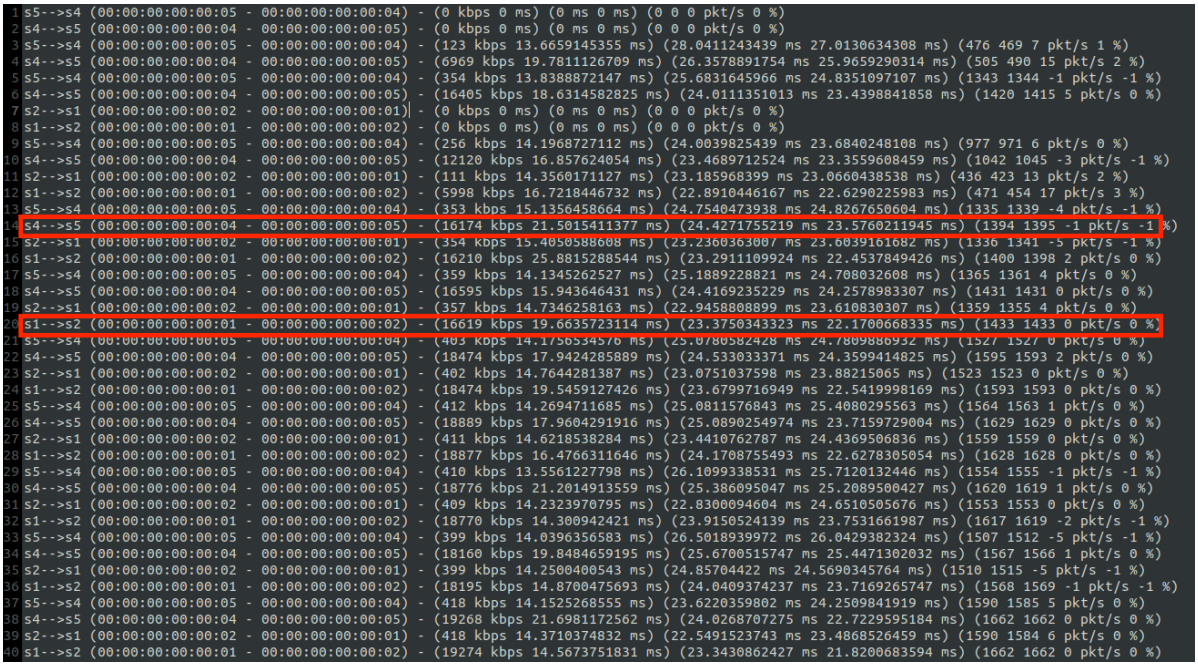


Figure 2.7: Bandwidth 20 Mbps, Linear Topology, server h1 client h2, server h4 client h5

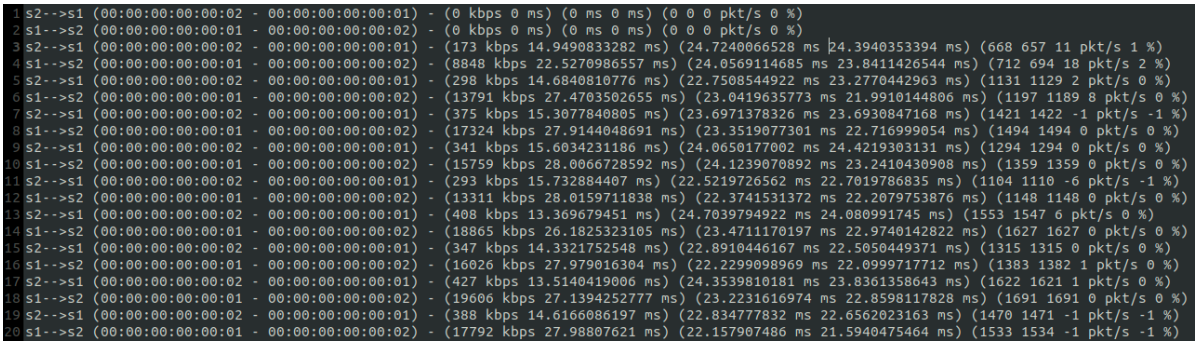


Figure 2.8: Bandwidth 20 Mbps, Ring Topology, server h1, client h2

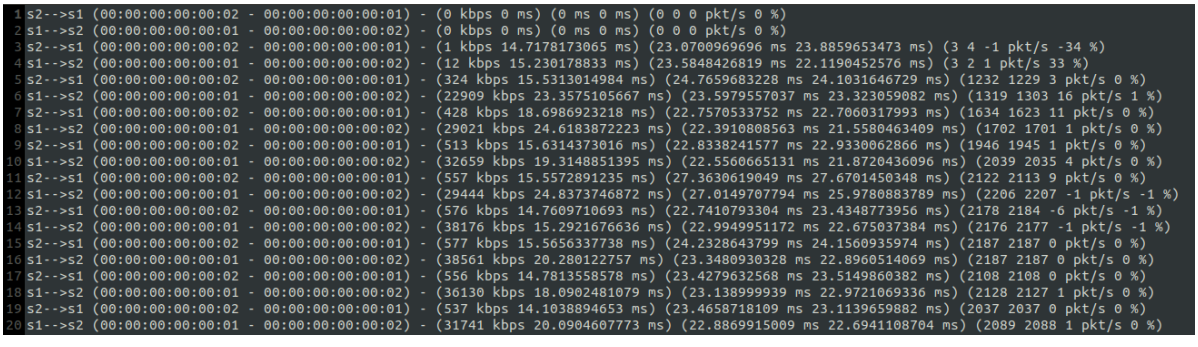


Figure 2.9: Bandwidth 40 Mbps, Linear Topology, server h1, client h2

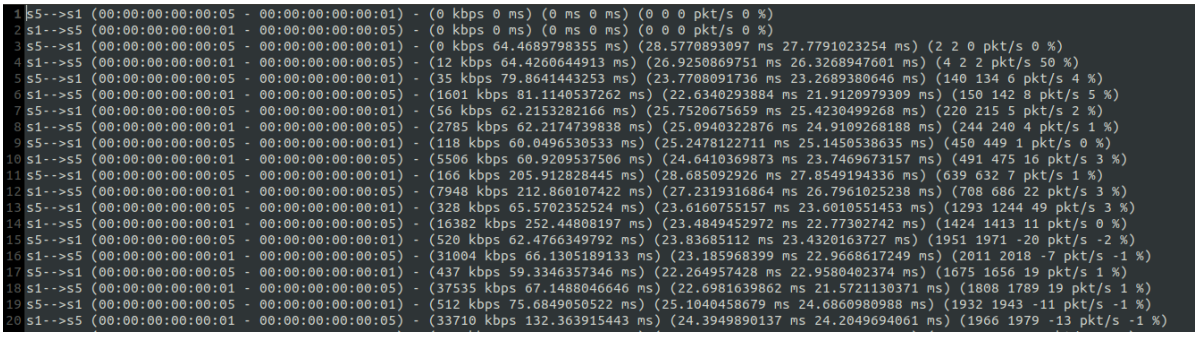


Figure 2.10: Bandwidth 40 Mbps, Linear Topology, server h1, client h5

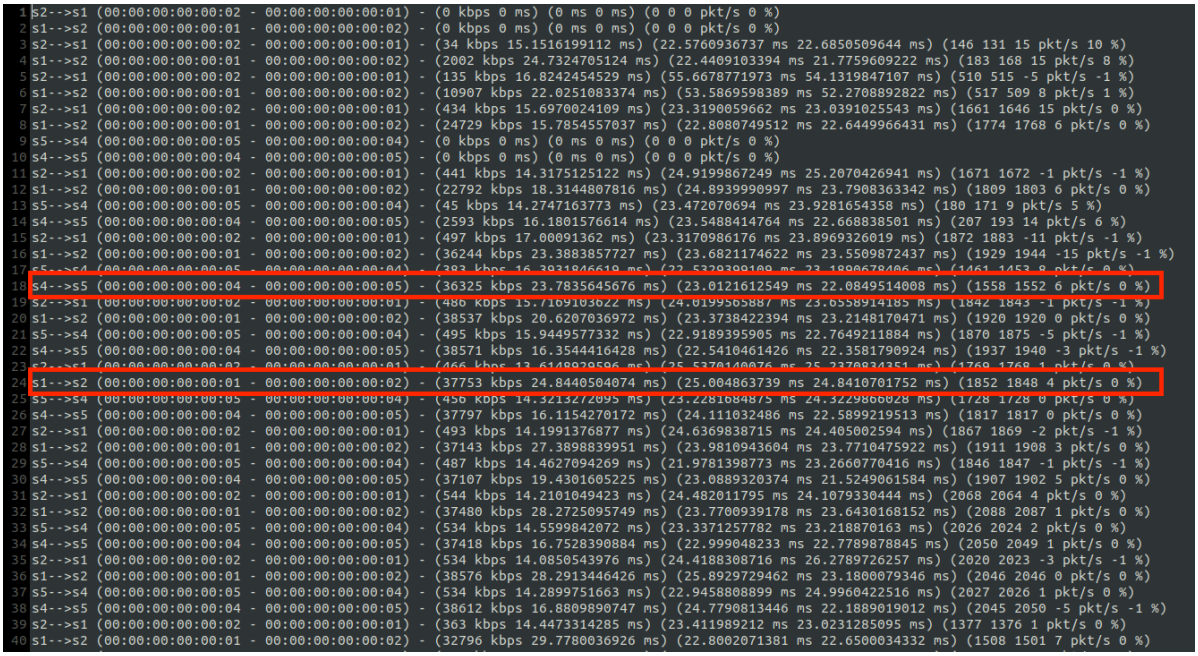


Figure 2.11: Bandwidth 40 Mbps, Linear Topology, server h1 client h2, server h4 client h5

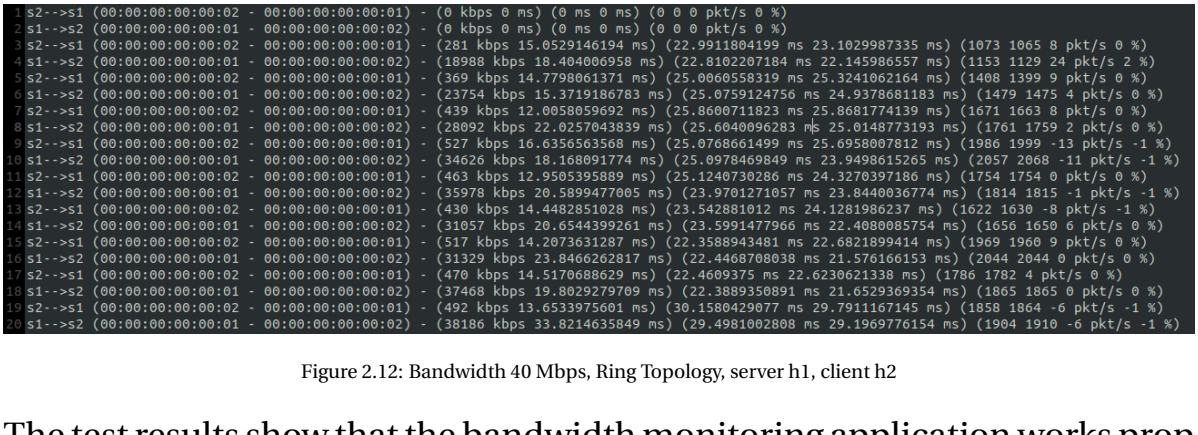


Figure 2.12: Bandwidth 40 Mbps, Ring Topology, server h1, client h2

The test results show that the bandwidth monitoring application works properly. Figure 2.5 and Figure 2.8 show that the bandwidth monitoring application is able to capture throughput of different network topology. Figure 2.7 and Figure 2.11 show that the bandwidth monitoring application is able to capture throughput for multiple flows in a network topology.

2.3. Delay Monitoring

2.3.1. Implementation

The Delay Monitoring applies the **active method** of QoS monitoring, which means it injects additional probe packets to monitor the network behaviour [7]. In order to calculate the delay of a data flow, we inject a delay probe packet to the network, the data path of the delay probe packet is the same as the data flow which is monitored. It is demonstrated in Figure 2.13. **t_probe** is the duration of the delay probe packet travelling in the data path. **t_1** and **t_3** are the duration of the delay probe packet traveling between controller and the source switch or the destination switch. Thus, the delay of the data flow between the source switch and the destination switch can be calculated as:

$$t\_delay = t\_probe - t\_1 - t\_3 \tag{2.1}$$

By processing the **PacketIn** event message, **t_probe** is able to obtained by the current time subtracting the time of the controller sent the delay probe packet.

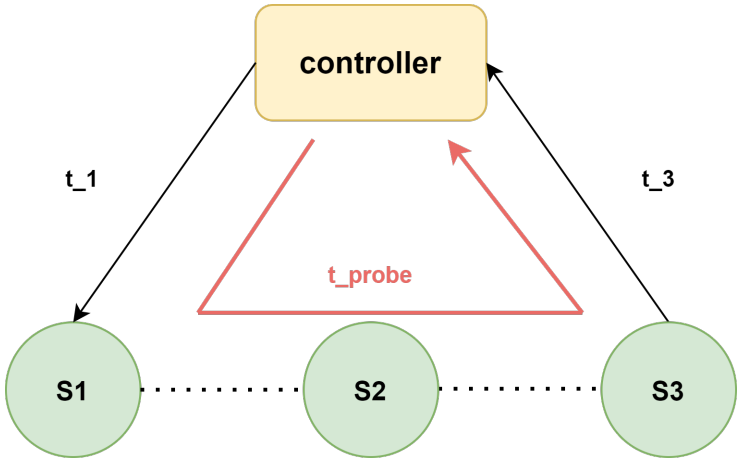


Figure 2.13: Delay Monitoring

t_1 or **t_3** can be regarded as half of the round trip time between the controller and the source switch or the destination switch. The round trip time can be obtained by the time different between the **OFPFlowStatsRequest** and the **OFPFlowStatsReply**. It is demonstrated in Figure 2.14. In order to record the time **t_a** of sending the request message to the aimed switch, a timestamp is sent along with **OFPFlowStatsRequest**. Thus, the delay can be calculated as:

$$t\_delay = t\_probe - 0.5 \times (t\_rtt\_src + t\_rtt\_dst) \tag{2.2}$$

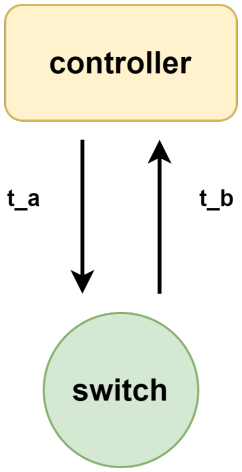


Figure 2.14: Round Trip Time

2.3.2. Testing

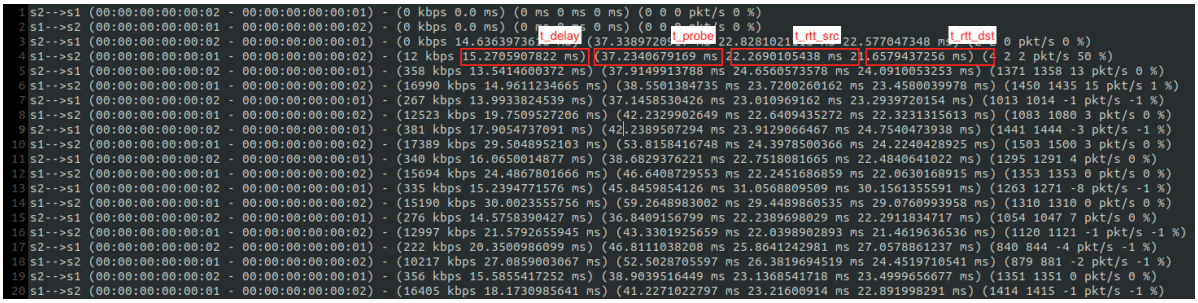
To verify the functionality of delay monitoring, we test the application with different delay settings and different topology configurations (**linear topology, complete graph topology**). Since **t_1** and **t_3** are too small to notice, we inject a delay of 10ms in the loopback interface in the network.

```
sudo tc qdisc add dev lo root netem delay 10ms
```

Table 2.3 gives an overview of all the tests for delay monitoring.

Figure	Topology	Delay (ms)	Server	Client
2.15	Linear	15	h1	h2
2.16	Linear	15	h1	h5
2.17	Linear	15	h1, h4	h2, h5
2.18	Complete Graph	15	h1	h5
2.19	Linear	30	h1	h2
2.20	Linear	30	h1	h5
2.21	Linear	30	h1, h4	h2, h5
2.22	Complete Graph	30	h1	h2

Table 2.3: Delay Monitoring Tests




```
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (0 kbps 14.636397363 ms) (37.338972095 ms) 22.8281021 ms) 22.577047348 ms) 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (12 kbps 15.2705907822 ms) (37.2340679169 ms 22.2690105438 ms 21.6579437256 ms) (4 2 2 pkt/s 50 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (358 kbps 13.5414600372 ms) (37.9149913788 ms 24.6560573578 ms 24.0910053253 ms) (1371 1358 13 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (16990 kbps 14.9611234665 ms) (38.5501384735 ms 23.7200260162 ms 23.4580039978 ms) (1450 1435 15 pkt/s 1 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (267 kbps 13.9933824539 ms) (37.1458530426 ms 23.010969162 ms 23.2939720154 ms) (1013 1014 -1 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (12523 kbps 19.7509527206 ms) (42.2329902649 ms 22.6409435272 ms 22.3231315613 ms) (1083 1080 3 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (381 kbps 17.9054737091 ms) (42.2389507294 ms 23.9129066467 ms 24.7540473938 ms) (1441 1444 -3 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (17389 kbps 29.5048952103 ms) (53.8158416748 ms 24.3978500366 ms 24.2240428925 ms) (1503 1500 3 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (340 kbps 16.0650014877 ms) (38.6829376221 ms 22.7518081665 ms 22.4840641022 ms) (1295 1291 4 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (15694 kbps 24.4867801666 ms) (46.6408729553 ms 22.2451686859 ms 22.0630168915 ms) (1353 1353 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (335 kbps 15.2394771576 ms) (45.8459854126 ms 31.0568809509 ms 30.1561355591 ms) (1263 1271 -8 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (15190 kbps 30.0023555756 ms) (59.2648983002 ms 29.4489860535 ms 29.0760993958 ms) (1310 1310 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (276 kbps 14.5758390427 ms) (36.8409156799 ms 22.2389698029 ms 22.2911834717 ms) (1054 1047 7 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (12997 kbps 21.5792655945 ms) (43.3301925659 ms 22.0398902893 ms 21.4619636536 ms) (1120 1121 -1 pkt/s -1 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (222 kbps 20.3500986099 ms) (46.8111038208 ms 25.8641242981 ms 27.0578861237 ms) (840 844 -4 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (10217 kbps 27.0859003067 ms) (52.5028705597 ms 26.3819694519 ms 24.4519710541 ms) (879 881 -2 pkt/s -1 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (356 kbps 15.5855417252 ms) (38.9039516449 ms 23.1368541718 ms 23.4999656677 ms) (1351 1351 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (16405 kbps 18.1730985641 ms) (41.2271022797 ms 23.21600914 ms 22.891998291 ms) (1414 1415 -1 pkt/s -1 %)
```

Figure 2.15: Delay 15 ms, Linear Topology, server h1, client h2

```
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (37 kbps 58.6359500885 ms) (84.0280056 ms 25.032043457 ms 25.7520675659 ms) (142 141 1 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (1577 kbps 63.2117986679 ms) (87.2402191162 ms 25.1059532166 ms 22.9508876801 ms) (161 140 21 pkt/s 13 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (82 kbps 67.1095848083 ms) (97.2139835358 ms 28.8038253784 ms 31.4049720764 ms) (323 314 9 pkt/s 2 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (3846 kbps 67.6110982895 ms) (96.1730480194 ms 30.5609703064 ms 26.5629291534 ms) (327 332 -5 pkt/s -2 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (133 kbps 59.5490932465 ms) (82.5960636139 ms 23.3929157257 ms 22.7010250092 ms) (523 507 16 pkt/s 3 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (6495 kbps 59.7360134125 ms) (81.9869041443 ms 22.3269462585 ms 22.1748352051 ms) (566 560 6 pkt/s 1 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (239 kbps 60.0615739822 ms) (82.7729701996 ms 22.8209495544 ms 22.6018428802 ms) (911 908 3 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (10947 kbps 62.2175931931 ms) (84.1970443726 ms 22.3019123077 ms 21.6569900513 ms) (977 944 33 pkt/s 3 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (387 kbps 60.2240562439 ms) (82.2179317474 ms 21.7859745026 ms 22.2017765045 ms) (1486 1466 20 pkt/s 1 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (17834 kbps 62.1314048767 ms) (84.0129852295 ms 21.9550132751 ms 21.8081474304 ms) (1539 1538 1 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (429 kbps 68.4759616852 ms) (95.263004303 ms 27.2591114044 ms 26.3149738312 ms) (1625 1625 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (19552 kbps 125.139832497 ms) (150.652885437 ms 25.6721973419 ms 25.3539085388 ms) (1688 1686 2 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (364 kbps 59.7101449966 ms) (82.043170929 ms 22.4440097809 ms 22.2220420837 ms) (1379 1380 -1 pkt/s -1 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (16660 kbps 67.8863525391 ms) (89.8098945618 ms 22.0029354095 ms 21.8441486359 ms) (1440 1437 3 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (368 kbps 61.5895986557 ms) (94.654083252 ms 33.2639217377 ms 32.8650474548 ms) (1396 1395 1 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (16814 kbps 130.929946899 ms) (163.44499588 ms 32.5899124146 ms 32.4401855469 ms) (1447 1450 -3 pkt/s -1 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (214 kbps 80.1331996918 ms) (104.215860367 ms 23.7522125244 ms 24.4131088257 ms) (805 813 -8 pkt/s -1 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (9844 kbps 181.109070778 ms) (205.053091049 ms 24.041891098 ms 23.8461494446 ms) (853 849 4 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (300 kbps 59.4201087952 ms) (82.4379920959 ms 23.2648849487 ms 22.7708816528 ms) (1149 1139 10 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (13833 kbps 69.2713260651 ms) (91.5329456329 ms 22.3610401154 ms 22.1621990204 ms) (1193 1193 0 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (425 kbps 60.0934028625 ms) (82.1528434753 ms 22.0658779144 ms 22.0530033112 ms) (1610 1613 -3 pkt/s -1 %)
```

Figure 2.16: Delay 15 ms, Linear Topology, server h1, client h5

```
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (88 kbps 14.2060518265 ms) (37.2960567474 ms 23.1881141663 ms 22.9918956757 ms) (341 336 5 pkt/s 1 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (6831 kbps 17.0130729675 ms) (39.2830371857 ms 22.4487781525 ms 22.0911502838 ms) (349 341 8 pkt/s 2 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (345 kbps 14.8887634277 ms) (39.0331745148 ms 24.2788791656 ms 24.0099430084 ms) (1316 1309 7 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (17204 kbps 15.7111883163 ms) (39.3011569977 ms 23.7128734589 ms 23.4670639038 ms) (1398 1390 8 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (170 kbps 14.8453712463 ms) (38.2008552551 ms 23.227930069 ms 23.4830379486 ms) (656 644 12 pkt/s 1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (11767 kbps 20.0130939484 ms) (42.9470539093 ms 23.2419967651 ms 22.6259231567 ms) (694 674 20 pkt/s 2 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (426 kbps 16.5414810181 ms) (42.1750545502 ms 25.3701210022 ms 25.897026062 ms) (1614 1614 0 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (19564 kbps 17.6305770874 ms) (42.4211025238 ms 25.505065918 ms 24.0759849548 ms) (1687 1687 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (426 kbps 15.9820318222 ms) (40.4510498047 ms 24.8019695282 ms 24.1360664368 ms) (1618 1617 1 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (19570 kbps 22.770524025 ms) (46.4661121368 ms 23.805141449 ms 23.5860347748 ms) (1687 1688 -1 pkt/s -1 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (408 kbps 13.3943557739 ms) (38.6168956757 ms 25.5069732666 ms 24.9381065369 ms) (1548 1548 0 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (18776 kbps 14.7250890732 ms) (38.7349128723 ms 24.0747928619 ms 23.9448547363 ms) (1620 1619 1 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (408 kbps 14.4256353378 ms) (37.3320579529 ms 22.6619243622 ms 23.1509208679 ms) (1545 1546 -1 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (18812 kbps 22.5419998169 ms) (44.8000431061 ms 22.8550434113 ms 21.6610431671 ms) (1624 1622 2 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (423 kbps 16.716837883 ms) (41.6398048401 ms 24.2609977722 ms 25.584936142 ms) (1606 1605 1 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (19458 kbps 15.9913301468 ms) (41.2118434906 ms 25.3090858459 ms 25.1319408417 ms) (1678 1678 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (423 kbps 16.7634487152 ms) (40.7199859619 ms 23.2200622559 ms 24.6930122375 ms) (1605 1605 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (19434 kbps 21.9305753708 ms) (45.5150604248 ms 24.514913559 ms 22.6540565491 ms) (1678 1676 2 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (378 kbps 14.7850513458 ms) (38.3930206299 ms 23.74792099 ms 23.4680175781 ms) (1434 1435 -1 pkt/s -1 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (17401 kbps 22.3990678787 ms) (45.5830097198 ms 23.2479572296 ms 23.1199264526 ms) (1506 1502 4 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (377 kbps 15.4573917389 ms) (37.740945816 ms 21.8250751495 ms 22.7420330048 ms) (1429 1429 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (17454 kbps 21.9589471817 ms) (43.9369678497 ms 22.5510597229 ms 21.4049816132 ms) (1503 1505 -2 pkt/s -1 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (425 kbps 14.0203237534 ms) (37.9159450531 ms 24.1501331329 ms 23.6411094666 ms) (1613 1613 0 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (19517 kbps 22.8954553604 ms) (46.1640357971 ms 23.3170986176 ms 23.2200622559 ms) (1684 1683 1 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (424 kbps 14.2534971237 ms) (37.1038913727 ms 22.9179859161 ms 22.7828025818 ms) (1609 1609 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (19523 kbps 20.635008812 ms) (43.1439876556 ms 22.5908756256 ms 22.4270820618 ms) (1684 1684 0 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (424 kbps 16.5820121765 ms) (41.6278839111 ms 25.2418518066 ms 24.8498916626 ms) (1608 1608 0 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (19422 kbps 23.1119394302 ms) (47.4710464478 ms 24.4691371918 ms 24.2490768433 ms) (1675 1675 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (424 kbps 16.5755748749 ms) (40.344953537 ms 23.9789485931 ms 23.5598087311 ms) (1608 1607 1 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (19428 kbps 23.9536762238 ms) (47.247171402 ms 23.344039917 ms 23.2429504395 ms) (1676 1675 1 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (415 kbps 14.6812200546 ms) (40.1232242584 ms 25.9649753571 ms 24.9190330505 ms) (1573 1573 0 pkt/s 0 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (19090 kbps 21.3854312897 ms) (45.9849834442 ms 24.6140956879 ms 24.5850086212 ms) (1646 1646 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (416 kbps 15.7290697098 ms) (38.7511253357 ms 22.4659442902 ms 23.5781669617 ms) (1577 1579 -2 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (19049 kbps 26.9814729691 ms) (49.3040084839 ms 23.246049881 ms 21.3990211487 ms) (1649 1645 4 pkt/s 0 %)
s5-->s4 (00:00:00:00:00:05 - 00:00:00:00:00:04) - (412 kbps 14.7154331207 ms) (38.8329029083 ms 24.2331027985 ms 24.0018367767 ms) (1561 1562 -1 pkt/s -1 %)
s4-->s5 (00:00:00:00:00:04 - 00:00:00:00:00:05) - (18993 kbps 24.3549346924 ms) (48.0389595032 ms 23.7770080566 ms 23.5910415649 ms) (1655 1651 4 pkt/s 0 %)
```

Figure 2.17: Delay 15 ms, Linear Topology, server h1 client h2, server h4 client h5

```
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (0 kbps 14.5825147629 ms) (37.1611118317 ms 22.2630500793 ms 22.8941440582 ms) (2 2 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (12 kbps 14.6545171738 ms) (36.6780757904 ms 22.6709842682 ms 21.3761329651 ms) (4 2 2 pkt/s 50 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (336 kbps 16.9448852539 ms) (41.5439605713 ms 25.1650810242 ms 24.0330696106 ms) (1282 1274 8 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (15517 kbps 18.3786153793 ms) (41.0220623016 ms 22.7649211884 ms 22.5219726562 ms) (1354 1340 14 pkt/s 1 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (357 kbps 14.3107175827 ms) (37.1851921082 ms 23.0309963226 ms 22.7179527283 ms) (1357 1353 4 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (16417 kbps 16.828417778 ms) (39.2239093781 ms 22.4599838257 ms 22.3309993744 ms) (1416 1417 -1 pkt/s -1 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (342 kbps 14.0000581741 ms) (37.4100208282 ms 23.5619544983 ms 23.2579708099 ms) (1298 1298 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (15664 kbps 15.1644945145 ms) (38.1109714508 ms 23.0150222778 ms 22.8779315948 ms) (1352 1351 1 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (354 kbps 14.6640539169 ms) (37.486076355 ms 22.5710868835 ms 23.0729579926 ms) (1343 1342 1 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (16293 kbps 21.9779014587 ms) (44.6119308472 ms 22.7251052856 ms 22.5429534912 ms) (1407 1405 2 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (356 kbps 14.4299268723 ms) (36.7789268494 ms 22.4339962006 ms 22.2640037537 ms) (1351 1352 -1 pkt/s -1 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (16376 kbps 19.1799402237 ms) (41.0568714142 ms 21.9719409943 ms 21.7819213867 ms) (1413 1412 1 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (332 kbps 14.5921707153 ms) (38.3400917053 ms 23.9050388336 ms 23.5908031464 ms) (1259 1258 1 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (15315 kbps 23.0988264084 ms) (46.1518764496 ms 23.1719017029 ms 22.9341983795 ms) (1323 1321 2 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (315 kbps 14.5806074142 ms) (38.1491184235 ms 23.5249996185 ms 23.6120223999 ms) (1190 1196 -6 pkt/s -1 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (14437 kbps 26.0541439056 ms) (48.7380027771 ms 23.0479240417 ms 22.3197937012 ms) (1249 1245 4 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (312 kbps 14.5099163055 ms) (38.0959510803 ms 23.5271453857 ms 23.6449241638 ms) (1188 1184 4 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (14337 kbps 30.0344228745 ms) (53.0998706818 ms 23.4038829803 ms 22.7270126343 ms) (1236 1236 0 pkt/s 0 %)
```

Figure 2.18: Delay 15 ms, Complete Graph Topology, server h1, client h5

```
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (10 kbps 30.5342674255 ms) (56.1242103577 ms 26.713848114 ms 24.4660377502 ms) (42 41 1 pkt/s 2 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (344 kbps 36.7068052292 ms) (59.8828792572 ms 23.4899520874 ms 22.8621959686 ms) (37 32 5 pkt/s 13 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (218 kbps 28.3906459808 ms) (53.2541275024 ms 25.0518321991 ms 24.6751308441 ms) (844 827 17 pkt/s 2 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (10662 kbps 34.6512794495 ms) (58.896780014 ms 24.3020057678 ms 24.1889953613 ms) (942 920 22 pkt/s 2 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (364 kbps 29.7803878784 ms) (52.2468090057 ms 22.1319198608 ms 22.8009223938 ms) (1388 1382 6 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (16915 kbps 32.9946279526 ms) (55.3441047668 ms 22.439956665 ms 22.2589969635 ms) (1464 1459 5 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (426 kbps 34.1933965683 ms) (56.7889213562 ms 22.7692127228 ms 22.421836853 ms) (1618 1617 1 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (19446 kbps 37.899851799 ms) (59.9598884583 ms 22.1440792084 ms 21.9759941101 ms) (1677 1677 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (327 kbps 29.6987295151 ms) (51.787853241 ms 21.9051837921 ms 22.2730636597 ms) (1240 1241 -1 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (15006 kbps 30.4338932037 ms) (52.3478984833 ms 21.9979286194 ms 21.8300819397 ms) (1291 1294 -3 pkt/s -1 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (399 kbps 29.6655893326 ms) (53.0600547791 ms 23.5288143158 ms 23.2601165771 ms) (1513 1513 0 pkt/s 0 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (18302 kbps 31.1110019684 ms) (53.6479949951 ms 22.8879451752 ms 22.1860408783 ms) (1580 1578 2 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (333 kbps 29.8565626144 ms) (53.1160831451 ms 22.8381156921 ms 23.6809253693 ms) (1248 1264 -16 pkt/s -2 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (15107 kbps 31.907916069 ms) (55.2179813385 ms 23.4110355377 ms 23.2090950012 ms) (1303 1303 0 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (393 kbps 29.3874740601 ms) (52.6578426361 ms 23.1788158417 ms 23.3619213104 ms) (1508 1491 17 pkt/s 1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (18320 kbps 33.3434343338 ms) (56.0100078583 ms 22.7580070496 ms 22.5751399994 ms) (1582 1581 1 pkt/s 0 %)
s2-->s1 (00:00:00:00:00:02 - 00:00:00:00:00:01) - (379 kbps 29.5193195343 ms) (52.7968406677 ms 23.2951641083 ms 23.2598781586 ms) (1437 1438 -1 pkt/s -1 %)
s1-->s2 (00:00:00:00:00:01 - 00:00:00:00:00:02) - (17395 kbps 37.5180244446 ms) (60.1968765259 ms 22.9778289795 ms 22.3798751831 ms) (1504 1500 4 pkt/s 0 %)
```

Figure 2.19: Delay 30 ms, Linear Topology, server h1, client h2

```
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (0 kbps 0.0 ms) (0 ms 0 ms 0 ms) (0 0 0 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (0 kbps 155.003070831 ms) (192.532062531 ms 37.8310680389 ms 37.2269153595 ms) (2 2 0 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (0 kbps 171.085000038 ms) (207.673072815 ms 36.6430282593 ms 36.5331172943 ms) (4 0 4 pkt/s 100 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (20 kbps 124.999642372 ms) (150.591135025 ms 25.7019996643 ms 25.4809856415 ms) (78 76 2 pkt/s 2 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (854 kbps 148.248076439 ms) (172.898054123 ms 25.0520706177 ms 24.2478847504 ms) (88 76 12 pkt/s 13 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (20 kbps 124.755620956 ms) (156.602144241 ms 32.1819782257 ms 31.5110683441 ms) (84 76 8 pkt/s 9 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (959 kbps 125.008940697 ms) (155.86400032 ms 31.0101509094 ms 30.699968338 ms) (78 82 -4 pkt/s -6 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (44 kbps 123.77512455 ms) (148.621082306 ms 24.8529911041 ms 24.838924408 ms) (176 167 9 pkt/s 5 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (2169 kbps 123.881220818 ms) (148.013830185 ms 24.2691040039 ms 23.9961147308 ms) (192 187 5 pkt/s 2 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (95 kbps 119.228839874 ms) (145.794868469 ms 26.5998840332 ms 26.5321731567 ms) (355 360 -5 pkt/s -2 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (4368 kbps 121.755480766 ms) (146.917819977 ms 25.8917808533 ms 24.4328975677 ms) (403 377 26 pkt/s 6 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (139 kbps 126.641511917 ms) (151.432037354 ms 24.973154068 ms 24.6078968048 ms) (549 528 21 pkt/s 3 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (6786 kbps 128.89456749 ms) (152.924060822 ms 24.1639614105 ms 23.8950252533 ms) (630 585 45 pkt/s 7 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (262 kbps 120.239853859 ms) (143.507003784 ms 23.4951972961 ms 23.0391025543 ms) (982 994 -12 pkt/s -2 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (12126 kbps 125.552535057 ms) (147.948980331 ms 22.5458145142 ms 22.2470760345 ms) (1066 1046 20 pkt/s 1 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (332 kbps 121.443986893 ms) (147.933006287 ms 26.7219543457 ms 26.2560844421 ms) (1263 1261 2 pkt/s 0 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (15214 kbps 151.877403259 ms) (177.516222 ms 25.741815567 ms 25.5358219147 ms) (1313 1312 1 pkt/s 0 %)
s5-->s1 (00:00:00:00:00:05 - 00:00:00:00:00:01) - (364 kbps 119.495391846 ms) (141.790866852 ms 22.0458507538 ms 22.5450992584 ms) (1454 1382 72 pkt/s 4 %)
s1-->s5 (00:00:00:00:00:01 - 00:00:00:00:00:05) - (17792 kbps 127.435445786 ms) (149.235963821 ms 22.3119258881 ms 21.2891101837 ms) (1535 1534 1 pkt/s 0 %)
```

Figure 2.20: Delay 30 ms, Linear Topology, server h1, client h5

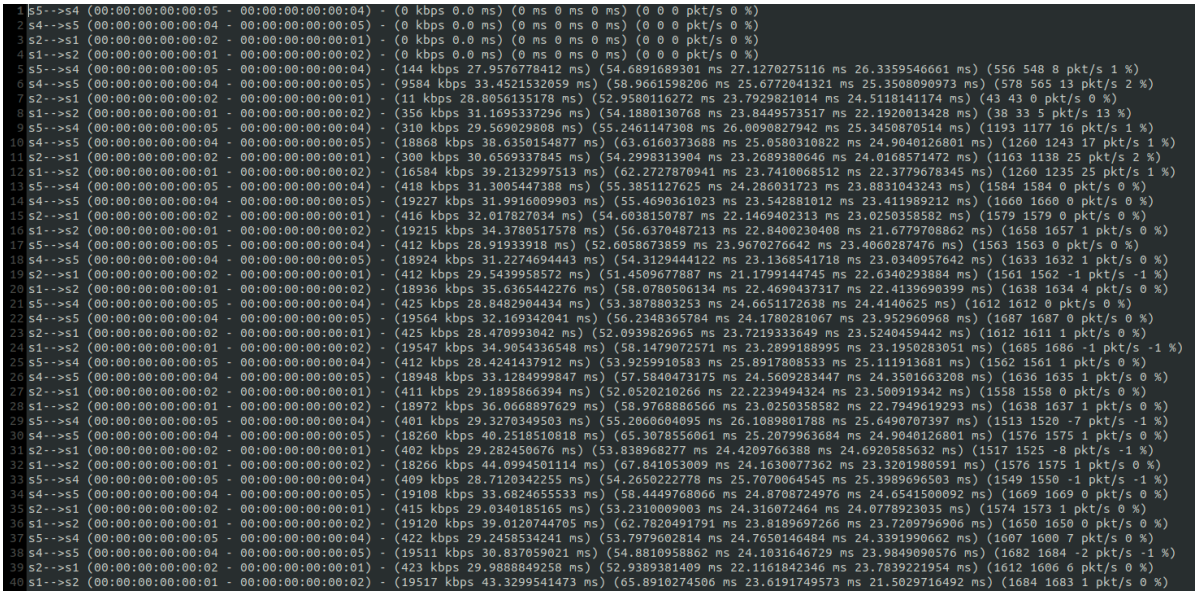


Figure 2.21: Delay 30 ms, Linear Topology, server h1 client h2, server h4 client h5

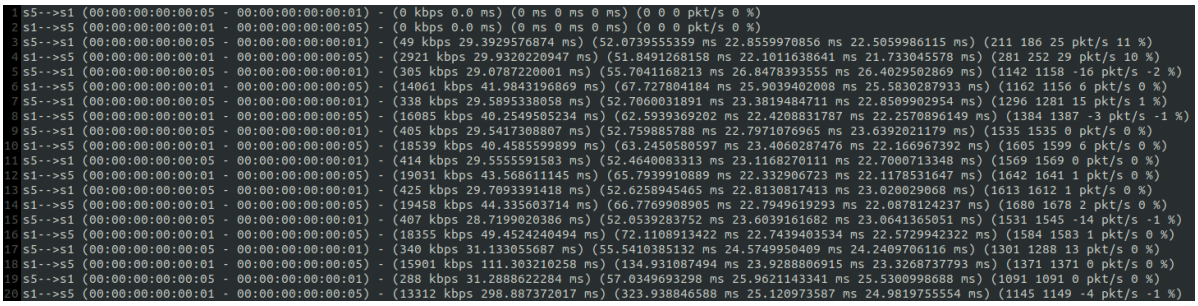


Figure 2.22: Delay 30 ms, Complete Graph Topology, server h1, client h5

Figure 2.17 and Figure 2.21 show that the monitoring application is able to capture the delay of multiple simultaneous flows. Figure 2.18 and Figure 2.22 show that the monitoring is able to work in a different topology, for example, a complete graph topology. Meanwhile, we notice that there are **some delay test results** different from our expectation:

1. The delay result is **a little smaller than** the expected delay result calculated by the delay configuration parameter times the links in the flow path.
2. The delay result is **larger than** the expected delay result calculated by the delay configuration parameter times the links in the flow path.

We assume that there are two reasons for the results:

1. The overhead caused by the Linux kernel.
2. The intermediate variables for calculating the delay are stacked in the computing process.

For the first reason, it can lead to a larger delay result. For the second reason, it can lead to an inaccurate delay result, either larger or smaller. In

the code section presented in Figure 2.23 of OpenFlow.py below, it requests statistics of the source switch and the destination switch from the OpenFlow controller and sends the latency probe packets by the OpenFlow controller to capture **t_probe**. The statistics for monitored paths are stored in the corresponding flow class instances presented in Figure 2.24. However, sometimes the update of variables in a flow class instances are not simultaneous due to the time processing the request message. Thus, there are some stacked intermediate variables in delay calculation.

```
140         def _monitor(self):
141                 """
142                 Main method for the monitoring actions.
143                 """
144                 while True:
145                         self.logger.info('Monitoring stats: num_flows:' + str(len(self.monitored_paths)))
146                         for m in self.monitored_paths:
147                                 print m
148                                 # send request statistics command to the source switch of the monitored flow
149                                 self.request_stats(m.sourceSwitch, m.src_mac, m.dst_mac, time.time())
150                                 # send request statistics command to the destination switch of the monitored flow
151                                 self.request_stats(m.destinationSwitch, m.src_mac, m.dst_mac, time.time())
152                                 self.send_latency_probe_packet(m.sourceSwitch, m.sourcePort, m.src_mac, m.dst_mac)
153                         hub.sleep(self.monitor_period)
```

Figure 2.23: Method **_monitor**

```
24 class Flow():
25         def __init__ (self, src, dst, sid, did, port):
26                 #values needed to identify the flow (src mac, dst mac, first and last switch)
27                 self.destinationSwitch = did
28                 self.sourceSwitch = sid
29                 self.src_mac = src
30                 self.dst_mac = dst
31                 #port from the first switch specifing the port to which the probe packet will be sent out to
32                 self.sourcePort = port
33                 self.delay = 0  #delay of the probe packet
34                 self.throughput = 0
35                 self.bytes_last = 0
36                 ##########################################################################################
37                 self.rtt_s_c_src = 0 #the round trip time between the source switch and the controller
38                 self.rtt_s_c_dst = 0 #the round trip time between the destination switch and the controller
39
40                 self.packet_src_last = 0
41                 self.packet_dst_last = 0
42                 ##########################################################################################
43                 self.packet_src = 0 #the total packet sent by the source switch
44                 self.packet_dst = 0 #the total packet received by the dst switch
45
46                 self.packet_loss = 0 #packet loss
47
48                 self.src_timestamp = 0 #record the current timestamp of the request message of source switch
49                 self.dst_timestamp = 0 #record the current timestamp of the request message of dst switch
50
51                 self.monitor_period = 0 #record the time interval for monitoring
```

Figure 2.24: Class **flow**

2.4. Loss Monitoring

2.4.1. Implementation

The loss monitoring is similar to the implementation of bandwidth monitoring. The loss of the data flow in this project is calculated as:

$$packet\_loss = (packet\_sent - packet\_received)/t\_int \tag{2.3}$$

The counts of packets sent by the source switch and the counts of packets sent by the destination switch can be obtained by the **OFPFlowStatsRequest**. It is demonstrated in the Figure 2.25.

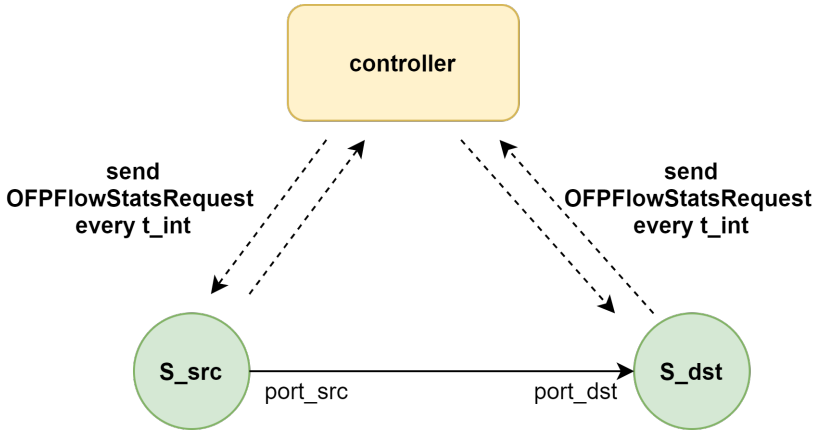


Figure 2.25: Loss Monitoring

2.4.2. Testing

To verify the functionality of loss monitoring, we test the application with different loss settings and different protocols (TCP, UDP). An example command is shown below:

```
sudo mn --mac --arp --topo=linear,5 --controller=remote \
--switch=ovs,protocol=OpenFlow13 \
--links=tc,bw=20,delay=15ms,loss=5
```

Table 2.4 gives an overview of all the tests for loss monitoring.

Figure	Topology	Loss Rate (%)	Server	Client	Protocol
2.26	Linear	0	h1	h2	tcp
2.27	Linear	0	h1	h2	udp
2.28	Linear	5	h1	h2	tcp
2.29	Linear	5	h1	h2	udp
2.30	Complete Graph	0	h1	h5	udp
2.31	Complete Graph	5	h1	h5	udp

Table 2.4: Loss Monitoring Tests

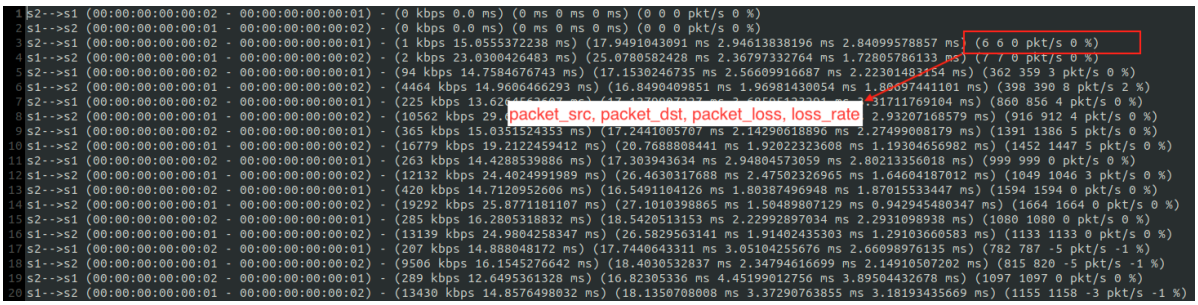


Figure 2.26: Loss 0 %, Linear Topology, tcp, server h1, client h2

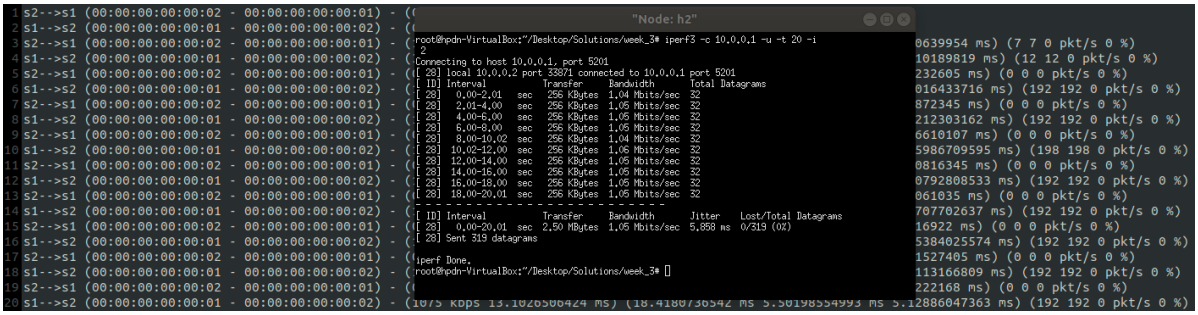


Figure 2.27: Loss 0 %, Linear Topology, udp, server h1, client h2

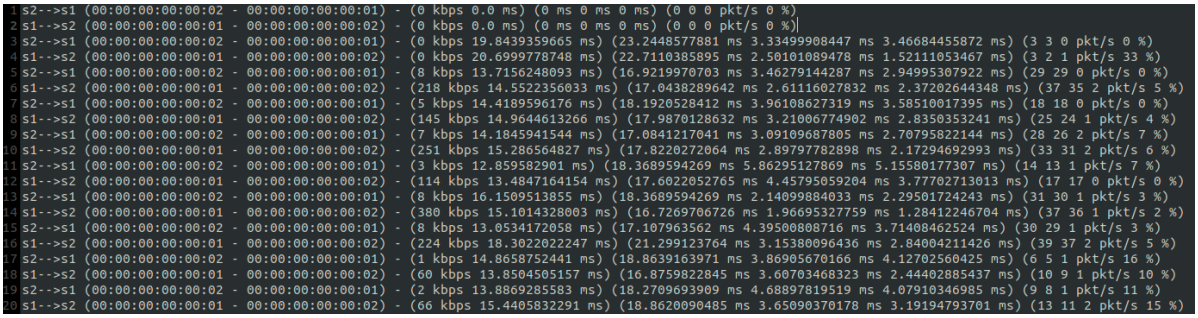


Figure 2.28: Loss 5 %, Linear Topology, tcp, server h1, client h2

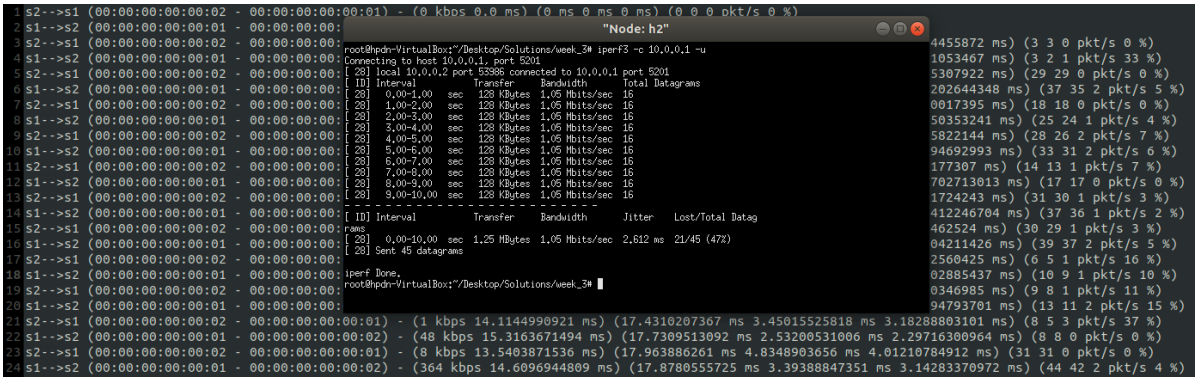


Figure 2.29: Loss 5 %, Linear Topology, udp, server h1, client h2

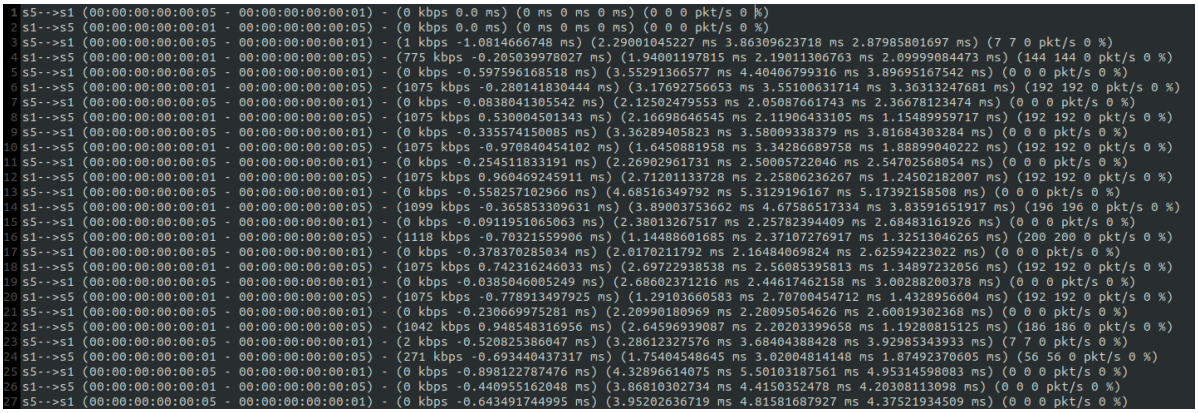


Figure 2.30: Loss 0 %, Complete Graph, udp, server h1, client h2

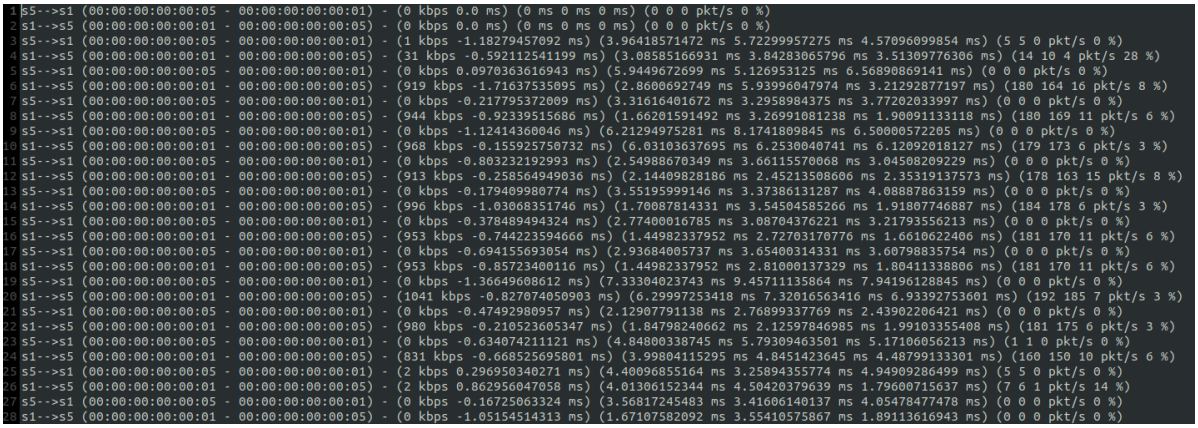


Figure 2.31: Loss 5 %, Complete Graph, udp, server h1, client h2

Figure 2.26 and Figure 2.28 compare test results of different loss rate configurations (0%, 5%)in TCP connection. Figure 2.29 to Figure 2.31 compare test results of different loss rate configurations and different topology configurations in UDP connection.

From the test results, we have noticed that some abnormal test results:

1. Negative loss rate: For example, in Figure 2.26, the configured loss rate is 0%, however the tested loss rate is below 0 for some test results.
2. Inaccurate loss rate: For example, in Figure 2.29, the configured loss rate is 5%, however, the tested loss rate is different from 5% for some test results.

We assume the reason for those abnormal results are caused by the reply statistics from the flow request messages. Once the **Monitoring** instance sends two flow request messages, two intermediate variables (**packet_src**, **packet_dst**) are returned by the reply message. **packet_src** is the number of packets sent by the server in the flow, **packet_dst** is the number of packets received by the client in the flow. However, when the **packet_dst** is returned by the reply message, there might exist some packets which haven't reached the destination switch.

2.5. Comparison

In this section, we will compare the advantages and disadvantages of an Active approach.

2.5.1. Advantages of Active approach

- It is less complex compared to passive approach, which gives real-time network statistics.
- This approach helps in providing network insights for a specified region.
- The data can be retrieved easily by polling the first and last switch.

2.5.2. Disadvantages of Active approach

- This approach does not measure real-time traffic delay, packet loss or bandwidth.
- It uses periodic polling is used, which gives less accurate results compared to adaptive polling.
- The probe packets generates additional traffic which can take up additional bandwidth and storage.

3

P4 Monitoring Implementation & Testing

3.1. P4 Monitoring

P4 (“Programming Protocol-independent Packet Processors”) is a language which specifies how packets are processed by the data plane of any switch, router or network interface card. In P4 also, there is a strict separation of data plane and control plane. These specifications can be installed on a switch and modified later. The operator can define how the packets are to be processed. In this way, P4 makes data plane flexible. The P4 monitoring implementation collects various throughput, delay and packet loss statistics.

This implementation of delay, bandwidth and loss monitoring is developed based on P4(v1.6). We will be making use of the provided virtual machine with the initial packet in packet out implementation of P4. In the report, we will mention, explain and screenshot most of the code we have added ourselves. For **delay monitoring**, we make uses of the ability to program the data plane to add information containing headers into the packets which in the end is forwarded to the controller and is extracted and processed by a monitoring thread. For **bandwidth monitoring** we will make use of the triple color meters and assign color rates in the central controller. For the **loss monitoring** we will use the integrated registers to contain the data which is sent to a monitoring thread in the controller. These three functionalities will be tested on a linear topology which consist of two hosts "h1", "h2" and three different switches "s1", "s2" and "s3", which are portrayed in Figure 3.1. Please notice that the controller is also portrayed and is part of the topology.

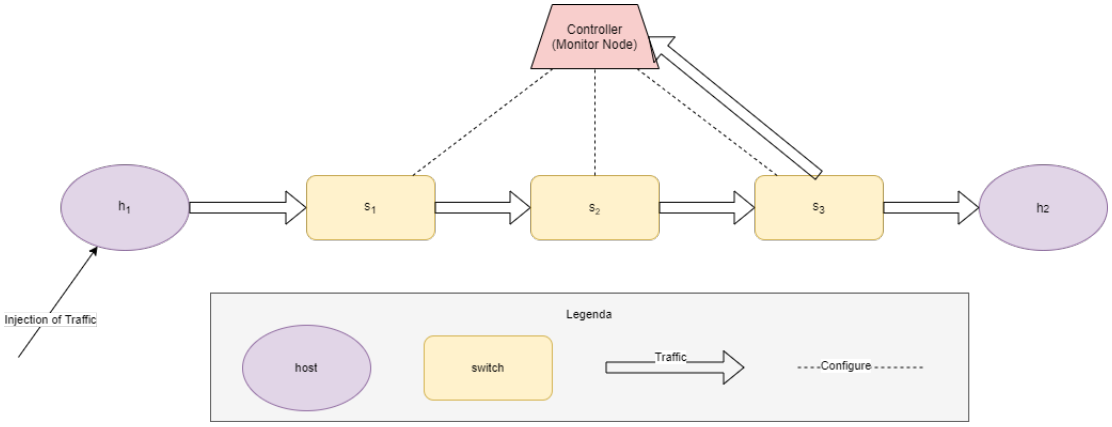


Figure 3.1: P4 testing topology 1

3.2. Diagnostic Node & Cloning

Before we begin with monitoring statistics from the network, a node is chosen to be the monitor node where monitoring information arrives and can be displayed. We have chosen the controller for various reasons: The controller is the "brain" of the network, therefore it is logical that statistics regarding the network should arrive at the controller. Another reason is, we can easily take advantage of the already existing implementation of packet in | packet out to forward packages containing information towards controller.

One of the most important thing to keep in mind is that, we need to implement this in such a way that the network traffic is unaffected. We choose to clone all packets containing network information from the final switch s3 before reaching destination h2 and send them towards controller. Imagine that network traffic from h1 to h2 would operate as normal and s3 is a "spy" node where all packets are forwarded towards the controller. Another approach could have been letting all the switches in the network clone and forward all packets towards the controller, but this approach is very impractical when it comes to a larger network where there are a lot of switches. With this approach, the controller would receive in a network of n nodes, per one single packet from h1 to h2, another extra n amount of cloned packets.

In order to perform cloning, we have to start a cloning session. This was done by initializing a cloning session in the controller.py[1] with the help for simple_switch_CLI. By default, SimpleSwitchGrpc do not support runtime_CLI, which needs to be enabled by compiling grpc switches with Thrift support [4].

After compiling the switch with Thrift support, we ran "netstat -tupnl" and got the following result, indicating success and all the switches are listening on their thrift ports.

[1] controller.py is the file that contains the controller codes

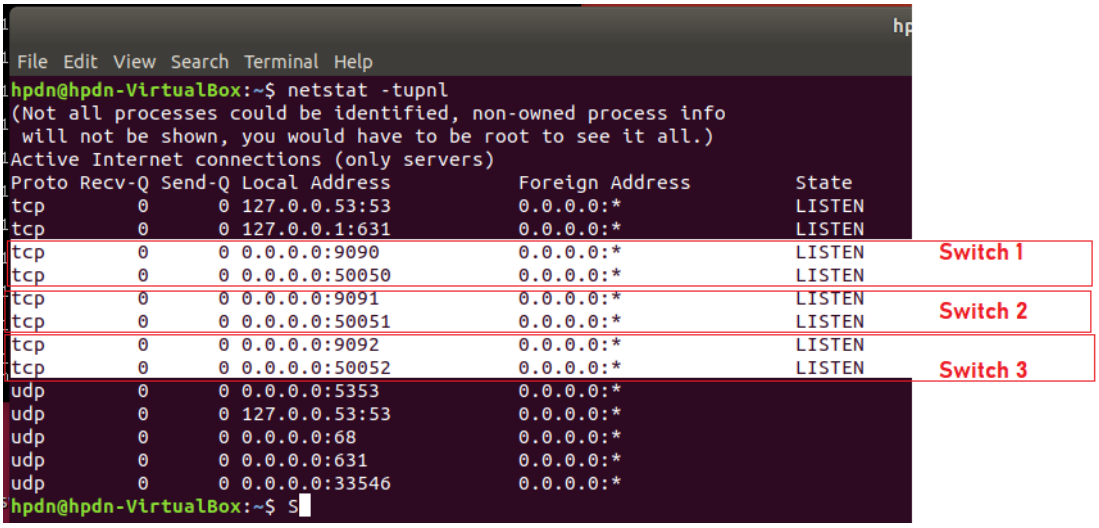


Figure 3.2: All thrift ports are listening

Now we set up the cloning. First, we used "Popen" from the subprocess module in the controller code to connect to switch 3 (which has thrift port of 9092) and created a mirroring session with the mirror id 2 (This was randomly chosen) and the egress port 255, which is the port towards the controller.

```
# Set up clone session for the final switch
print "\nSet up clone session from final switch towards controller"
s3_CLI = subprocess.Popen(["python behavioral-model/targets/simple_switch/simple_switch_CLI.in \
                --thrift-port 9092"], shell = True, stdout = subprocess.PIPE, stdin = subprocess.PIPE)
s3_CLI.communicate(input = "mirroring_add 2 255")[0]
```

Figure 3.3: Setting up the a clone session in controller.py

Next, in the p4 code switch_final.p4[2] of the final switch, we used clone3 function to enable Ingress to Egress cloning [1], which create a new copy of the current packet from the end of of the Ingress pipeline.

If the cloning is successful, we should receive ping replies from host 2 as well as receiving the new packets in the controller. See figure 3.4.

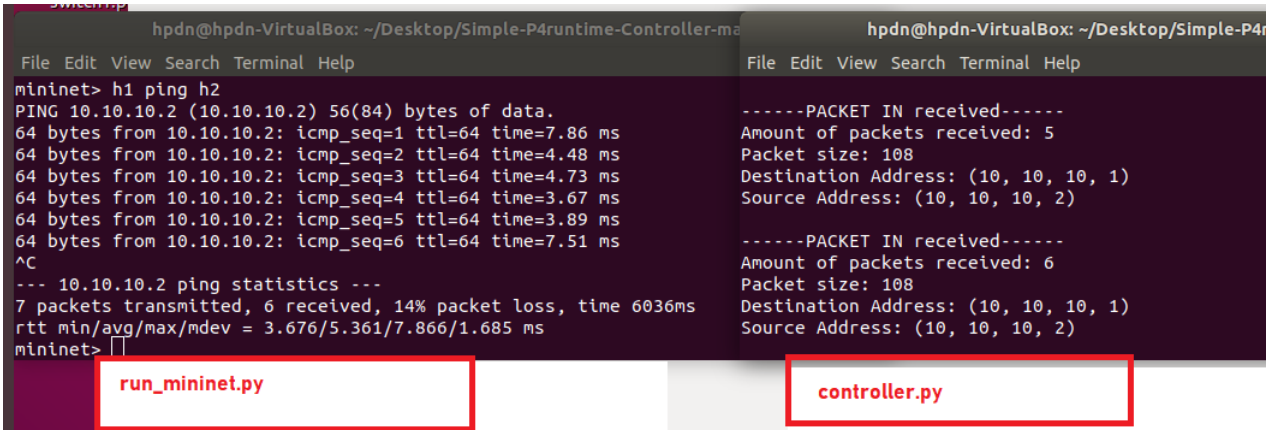


Figure 3.4: Cloning successful. Both Host 2 and Controller has received the packets

[2]switch_final.p4 is the file containing code for the final switch in the path which is switch 3

3.3. Delay Monitoring

3.3.1. Implementation

The P4 delay monitoring is achieved by modifying the program of every switches to add an extra header called "latency_t" into each packet as instructed in the P4 lecture. The goal is to make sure that every hop in the network, the statistics can be added to an extra header. In this case, we would like to measure the queue delay of each switch, i.e. how long does a particular switch takes to process an average packet. This is essentials in the future in order to find out where in the network is the bottle neck and route traffic accordingly.

Within every P4 switch code in the egress we add a new action called "add_latency_information()" which is called every single time in the "apply" section of the egress. In this method we set the latency header to be valid as well as adding queuing delay of that particular switch by using information from the standard_metadata. We assign "queuing_delay" attribute from the header to be the "deq_timedelta" from the metadata, which is the time, in microseconds, that the current packet spent in the queue [5]. This code for the switches can be seen at switch.p4 [3]. Do note that "deq_timedelta" will be 0 if assigned in the ingress, since the queuing take places between the ingress and egress. Therefore the correct way to get it is assign it in the egress.

The final switch on the path toward destination has the following unique function: besides adding delay information header like the other switches, it also clone the packet and forward them to both the original destination as well as the controller, as discussed in 3.2. A copy is made from the original packet which is forwarded to port 255 towards the controller. In the egress, we have implemented an if else case where if it is not a cloned packet, it is meant for the original recipients and all the extra latency headers should be removed. We do so by setting the headers stacks to invalid. The "latency_t" headers from the previous hops are removed so that this copy can be accepted by the destination as the "normal" packet. The destination will respond to the packet as if it was a normal packet without. This code for the final switch can be seen at "switch_final.p4"

[3] switch.p4 is the file containing code for the all switch in the network, except the final node which is switch 3

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

   action add_latency_information() {
        hdr.lastlatency.setValid();
        hdr.lastlatency.queuing_delay = standard_metadata.deq_timedelta;
    }

    apply {
        //Add switch latency information header, but only one way
        if(meta.stats_enabled){
            add_latency_information();
        }

        if(standard_metadata.instance_type == PKT_INSTANCE_TYPE_INGRESS_CLONE){
            //If this packet is a clone towards controller, we want to keep the headers for diagnostic purposes
            //So we do nothing
        } else{
            //If this packet is not a clone we want to remove the latency headers, to forward to original recipents as correct packet
            //Remove latency headers, but only one way
            if(meta.stats_enabled){
                //HARDCODED Element: there are 2 switches so there should be 2 latency headers
                hdr.latency[0].setInvalid();
                hdr.latency[1].setInvalid();
                hdr.lastlatency.setInvalid();
            }
        }
    }
}
```

Figure 3.5: Removing the latency headers in the egress if going towards original recipients

Notice that we hard-coded the amount of headers to be removed in the else case. This is not a good practice and should be avoided in the future by adding an extra variable in the header to indicate the final latency header when parsing the header stacks. An example implementing header stack and its parsing correctly so that there is no need for hard coding can be seen from a routing example [6].

Also notice that the "add_latency_information()" is only added one way, i.e. when the flow has stats enabled.

On the other hand, the clone is sent to the controller though the packet-in/packet-out mechanism. This copy still contains the delay headers which is extracted by a thread is constantly running in the controller. For each packet according to the topology used, there will be 3 headers which belong to switch 1, 2 and 3 according to the order this packet went through in the network. These three headers are extracted and simply printed.

3.3.2. Testing

In order to test the delay monitoring, the first step is running a modified version of "UPDsend.py" provided by the course. This script can be seen at "LatencyTest_UDPsend.py" which will attempt to non-stop sending UDP packets with destination as Host 2. This script is run at Host 1 which can be accesed with "xterm h1" when running mininet.

```
#!/usr/bin/python

from scapy.all import *
import argparse
import random


def main():
    """
    """


def packet_with_seq_n(port):
    packet = Ether(src="00:00:00:00:00:01", dst="00:00:00:00:00:02")/IP(src="10.10.10.1", dst="10.10.10.2")/UDP(sport=80, dport=port)/"UDPLatencyTest"
    sendp(packet, iface="eth0")

if __name__ == "__main__":
    main()
    parser = argparse.ArgumentParser(description='Simple script that sends TCP packets to an interface using scapy package')
    parser.add_argument('--dst-port', help='Destination port', type=int, action="store",    default=90)
    args = parser.parse_args()


    while True: #Send forever
        packet_with_seq_n(int(args.dst_port))
```

Figure 3.6: Code from the file LatencyTest_UDPsend.py

After running this "forever UDP sending" script, we take a look at the normal queue delay at the 3 switches which is parsed from the extra headers and printed on the CMD of the controller.py every time a packet arrives at the controller. The delay (microsecond!) is usally in the range of 2 to 3 digits at most.

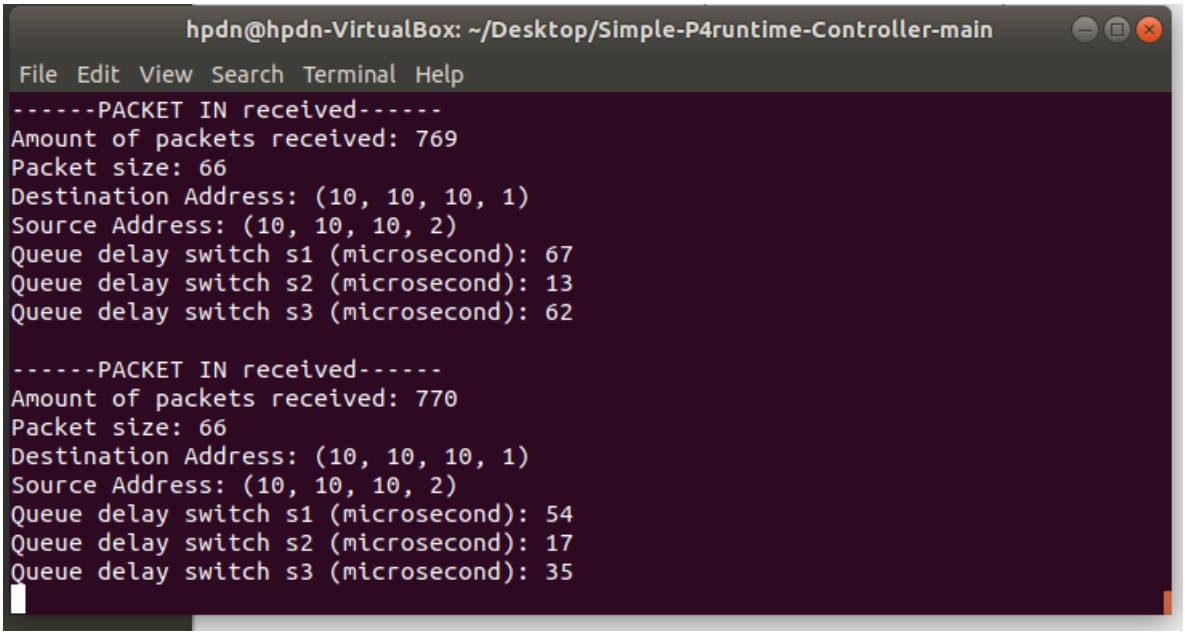


Figure 3.7: Queue delays in the range of 2 to 3 digits from three switches

Now we open up a simple_switch_CLI, connect to each of the switches 1, 2 and 3 with thrift ports 9090, 9091 and 9092 respectively and lower the queue rate with "set_queue_rate rate" command. See the following figures 3.8, 3.9 and 3.10 for the results.

```
hpdn@hpdn-VirtualBox:~/Desktop/Simple-P4runtime-Controller-main/behavioral-model
/targets/simple_switch$ python simple_switch_CLI.in --thrift-port 9090
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: set_queue_rate 20
RuntimeCmd:
```

hpdn@hpdn-VirtualBox: ~/Desktop/Simple-P4runtime-Controller-main

File Edit View Search Terminal Help

```
------PACKET IN received------
Amount of packets received: 13433
Packet size: 66
Destination Address: (10, 10, 10, 1)
Source Address: (10, 10, 10, 2)
Queue delay switch s1 (microsecond): 3104796
Queue delay switch s2 (microsecond): 50
Queue delay switch s3 (microsecond): 27

------PACKET IN received------
Amount of packets received: 13434
Packet size: 66
Destination Address: (10, 10, 10, 1)
Source Address: (10, 10, 10, 2)
Queue delay switch s1 (microsecond): 3100059
Queue delay switch s2 (microsecond): 105
Queue delay switch s3 (microsecond): 36
```

Figure 3.8: Delay increased after set queue rate of switch 1 to 20

```
hpdn@hpdn-VirtualBox:~/Desktop/Simple-P4runtime-Controller-main/behavioral-model
/targets/simple_switch$ python simple_switch_CLI.in --thrift-port 9091
Obtaining JSON from switch...
Done
Control utility for runtime P4 table manipulation
RuntimeCmd: set_queue_rate 20
RuntimeCmd:
```

hpdn@hpdn-VirtualBox: ~/Desktop/Simple-P4runtime-Controller-main

File Edit View Search Terminal Help

```
------PACKET IN received------
Amount of packets received: 15030
Packet size: 66
Destination Address: (10, 10, 10, 1)
Source Address: (10, 10, 10, 2)
Queue delay switch s1 (microsecond): 3164973
Queue delay switch s2 (microsecond): 469653
Queue delay switch s3 (microsecond): 107

------PACKET IN received------
Amount of packets received: 15031
Packet size: 66
Destination Address: (10, 10, 10, 1)
Source Address: (10, 10, 10, 2)
Queue delay switch s1 (microsecond): 3161705
Queue delay switch s2 (microsecond): 472571
Queue delay switch s3 (microsecond): 40
```

Figure 3.9: Delay switch 2 increased after set queue rate of switch 2 to 20

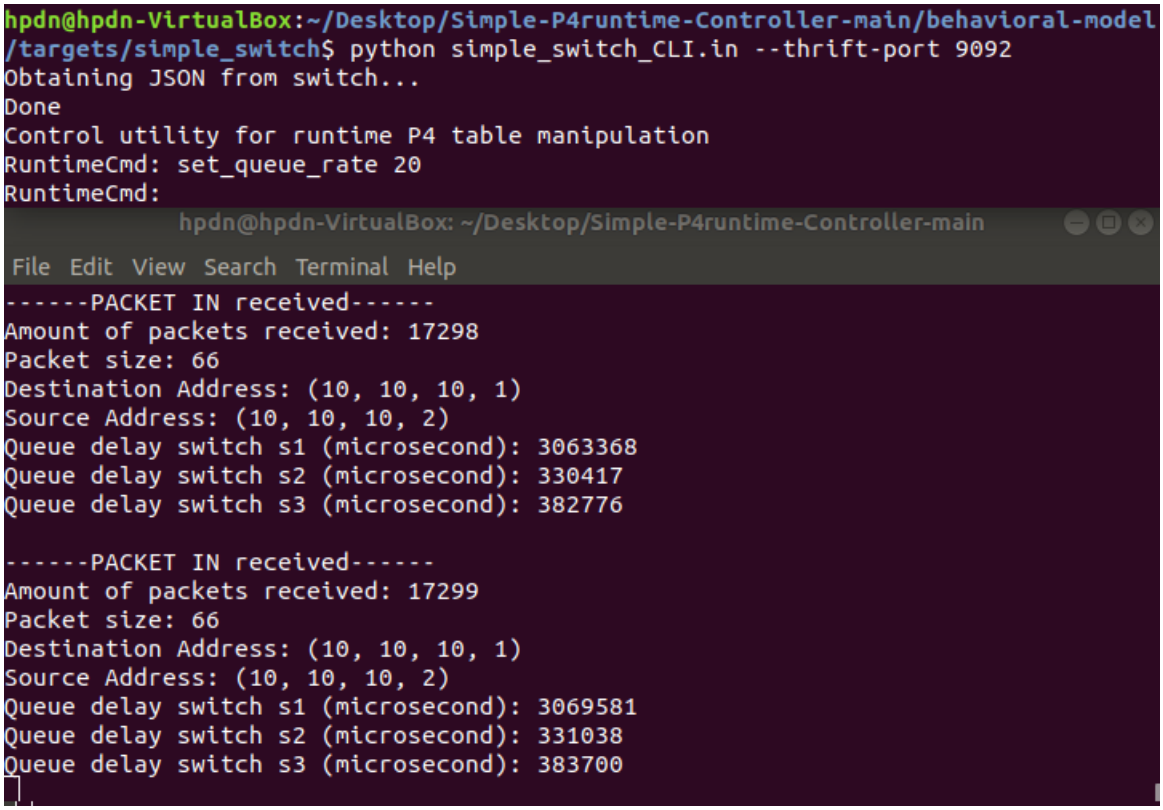


Figure 3.10: Delay switch 3 increased after set queue rate of switch 3 to 20

The delays increase is caused due to the rate of processing packets in the queues of the switches are lowered. This is reflected in the tests.

We noticed that the delays started to increase dramatically after setting the queue rate near the point of 25 packets per second rate. Above 30 packets per seconds there are no changes to the delays experienced.

3.4. Bandwidth Monitoring

3.4.1. Implementation

The P4 bandwidth monitoring is achieved by implementing a meter which is one of the extern object that is supported by P4. The point of a meter is to measure the rate of flow of packets through a switch and classify them into one of the assigned three categories automatically. The high flows are classified as red, medium flows as yellow and low flows as green. They are classified based on the strength of the flow relative to each other in a meter which is an array of multiple states of multiple flows.

We start with creating a meter in switch.p4 file which is called "bandwidth_meter" with size of 10 and choose to measure packets with "MeterType.packets".

```
/*************************************************************************
*************** I N G R E S S   P R O C E S S I N G   *******************
*************************************************************************/

control MyIngress(inout headers hdr,
                  inout metadata meta,
                  inout standard_metadata_t standard_metadata) {

    //meter definition
    meter(10, MeterType.packets) bandwidth_meter; //measuring based on packets
```

Figure 3.11: Meter initialization in switch.p4

On the headers we have also defined three different meter level representing green, yellow and red rate of packet flow. They are represented with 2 bit field "meter_level_t" as integer 0, 1 and 2 respectively, see 3.12 . We also added a variable "meter_level" into the meta data to hold the status of the flow.

```
//params for meter
typedef bit<2> meter_level_t;
const meter_level_t METER_COLOR_GREEN = 0;
const meter_level_t METER_COLOR_YELLOW = 1;
const meter_level_t METER_COLOR_RED = 2;
```

Figure 3.12: Meter colors initialization in switch.p4

In the ingress, we have implemented two different tables in order to make the meter work properly. The first table is called "flow_table" which has the function of classifying new packages to the right "index", which is used to identify one of the states of the meter. This is done by matching the packet's destination MAC address, i.e. the packets belong to the same flow/index/state of a meter if they have the same destination MAC address. We have implemented this table in such a way that in the controller, you can customize which switch to perform the bandwidth monitoring with the meter and which destination MAC address to keep track off.

The second table in the Ingress is called "flow_filter_table" which serves the purpose of filtering the packets by their meter level. For example, in the controller.py, we can install rules such that all packets belongs to a flow that has "become" red, i.e. become to large in the amount of incoming packets, to be dropped. For instance, if a packet comes in the "meta.meter_level" is 2, we can install the drop action.

```
//Table to keep tracks of different flow according to rules added in the controller.py
table flow_table{
    key = {
        hdr.ethernet.dstAddr: exact;
    }
    actions = {
        modify_flow_level;
        NoAction;
    }
    size = 1024;
    default_action = NoAction();
}

//Table to filter and drop flows according to rules added in the controller.py
table flow_filter_table{
    key = {
       meta.meter_level : exact;
    }
    actions = {
        drop;
        NoAction;
    }
    size = 16;
    default_action = NoAction();
}
```

Figure 3.13: Implementation of flow_table and flow_filter_table in switch.py

For the implementation of the meter and the tables, refer to the file "swit ch.p4". For the implementation of writing the rules into these tables, refer to the file "controller.py". In the controller, we only write methods/ rules for switch 1, which is to measure all traffic with destination mac adress to be host 2 and only drop level 1 packets, see 3.14. Since we have added tables to the switch code, we had to modify the "firmeware.p4info.txt" as well to include the new tables.

```
# Writing rules
# Forwarding rules
writeIpv4Rules(p4info_helper, sw_id=s1, dst_ip_addr="10.10.10.1", port = 1)
# withStats = True to only include latency header one waySS
writeIpv4Rules(p4info_helper, sw_id=s1, dst_ip_addr="10.10.10.2",  port = 2, withStats = True)

# Rule for bandwidth monitoring on switch 1
# Add which destination mac address to add a meter state to monitor
writeFlowTableRules(p4info_helper, sw_id=s1, dst_mac_addr="00:00:00:00:00:02")
# Add which at meter rules in order to drop yellow (level 1) packets
#writeFlowFilterTableRules(p4info_helper, sw_id=s1, drop_level= 1)

readTableRules(p4info_helper, s1)  #read all table rules
```

Figure 3.14: Meter assignement and bandwidth drop rules in controller.py

3.4.2. Testing

To test the bandwidth monitoring function, we start with running mininet as usual and then run controller.py to add rules to the switches. This time, start two XTERM windows of host 1 and host 2 with "xterm h1" and "xterm h2" respectively. On host 2 we start a UDP listen with "iperf -s -u -i 1" and on host 1 we sent UDP data with "iperf -c 10.10.10.2 -u -b 1M -l 1000". We are sending UDP datagrams from host 1 to host 2 with bandwidth of 1 megabit

per second and datagram size of 1000 bytes. See 3.15 below, you can see that the bandwidth towards host 2 was roughly 1 megabit per second as desired.

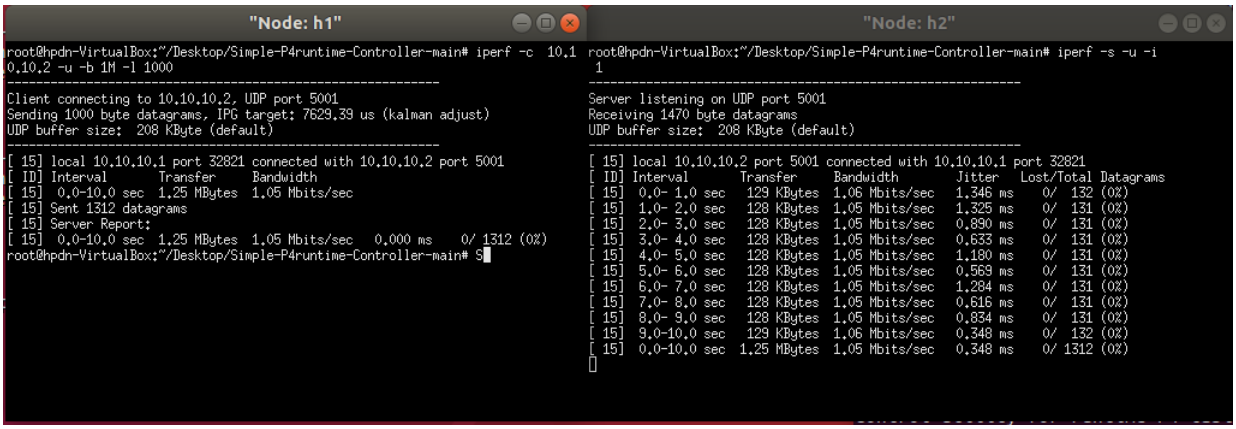


Figure 3.15: Meter assignement and bandwidth drop rules in controller.py

Now we will open up the simple_switch_CLI once again, connect with switch 1 and run the following command "meter_set_rates" bandwidth_meter 2 0.0001:1 0.0005:1". This means that we set the rate for the yellow color of the meter to be 100 packets per seconds.

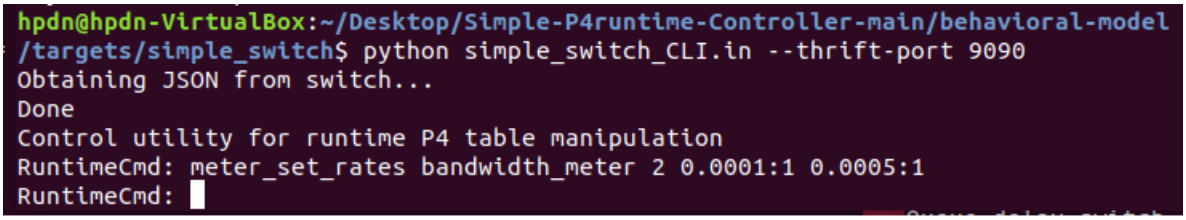


Figure 3.16: Setting rates for the meter in switch 1

Do remember from figure 3.14 that we have added rules to drop all packets that are yellow (drop_level = 1). So the maximum bandwidth achieved after dropping packets should be equal 100 packets/ second * 1000 bytes/ packet = 100000 bytes/ second. This is equal to 800 Kbits/second. The following figure shows the result of running the same iperf test several time again.

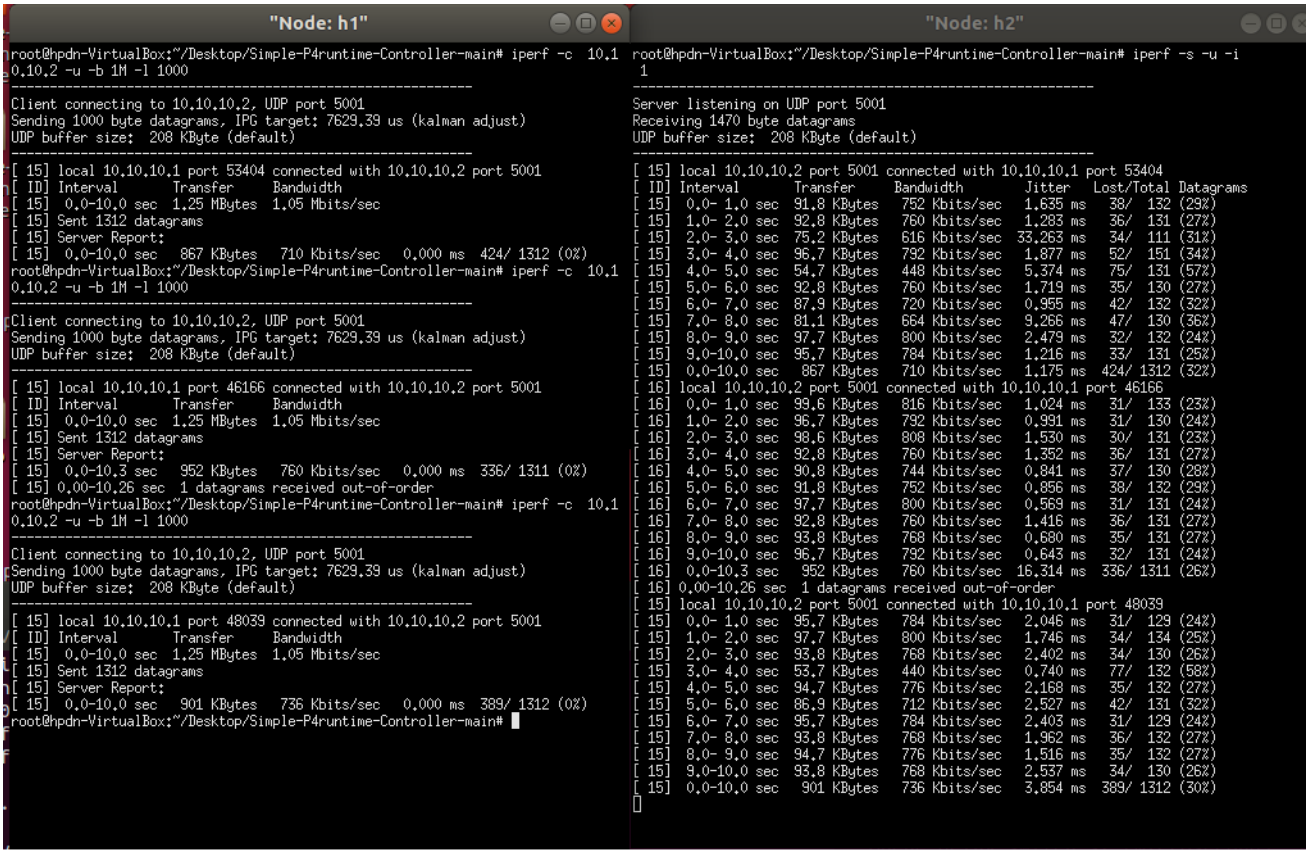


Figure 3.17: Result of rerunning the iperf test several times again

As you can see, the maximum bandwidth after setting the meter rates with dropping rules are always roughly 800 Kbits/second. This means the meter is working properly.

As a bonus, we disabled the rule to drop level 1 packets in the controller, rerun the controller and rerun the iperf test. Indeed from the result below you can see that the packets are not dropped anymore and the maximum bandwidth is back to roughly 1 Mbits/second.

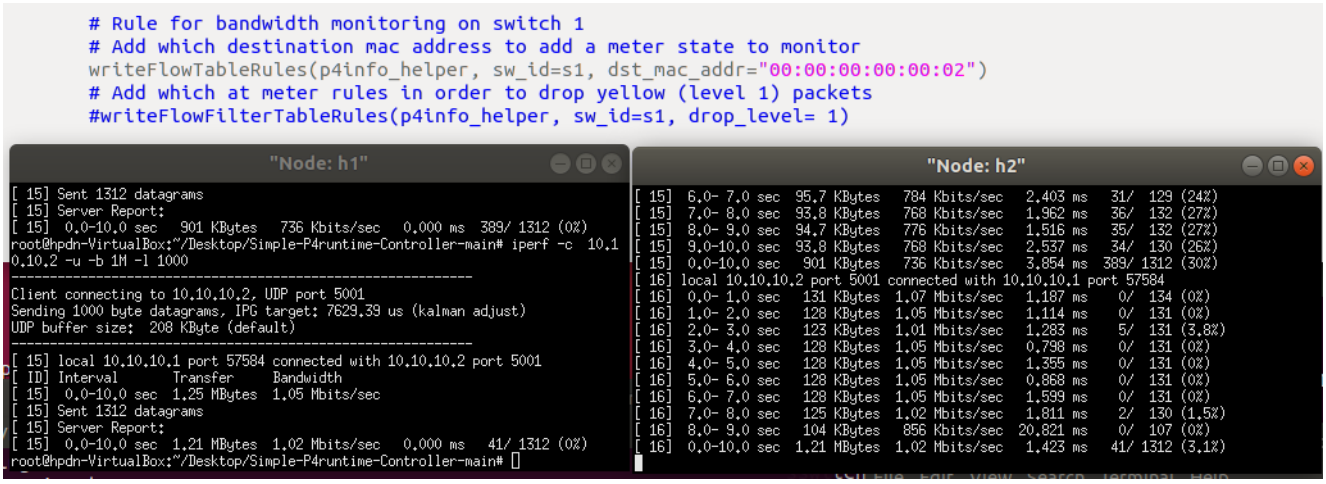


Figure 3.18: Bandwidth return back to normal after disable the meter dropping rule in controller.py

3.5. Loss Monitoring

3.5.1. Implementation

Loss monitoring was achieved with the help of using registers. In both switch.p4 and switch_final.p4, we have added a register called packet_counter. This register is used to log the amount of packets received based on the IPv4 identification header. The idea is that if a packet fragment consisting of 100 packets (which have the same IPv4 identification header) passes through a switch, the register with the index of that identification number will be incremented 100 times, if there are no packet loss at that switch. We have an action called update_counter which read the register at the index which is equal to IPv4 identification number of a packet and increment it. See figure 3.19.

```
control MyEgress(inout headers hdr,
                 inout metadata meta,
                 inout standard_metadata_t standard_metadata) {

    //register definition for loss monitoring
    register<bit<32>>(65536) packet_counter;

    action update_counter(bit<16> id){

        //First read the amount counted packets with this id
        bit<32> count;
        packet_counter.read(count, (bit<32>)id);

        //Increment by 1
        count = count + 1;
        //record the ID into the register
        packet_counter.write((bit<32>)id, count);
    }

       //action for delay calculation
    action add_latency_information() {
        hdr.latency.setValid();
        hdr.latency.queuing_delay = standard_metadata.deq_timedelta;
    }

    apply {
        update_counter(hdr.ipv4.identification);//update for loss monitoring
        if(meta.stats_enabled){
```

Figure 3.19: Implementation of register and method to update it in the switch.p4

In retrospect, this can also be done with a counter, but we have decided to keep register in case of further expansion of loss monitoring in the future.

3.5.2. Testing

To test loss monitoring, we would run from Host 1 a python file called LossTest_UDPsend.py. This file would send in total 1000 UDP packets with a payload consisting of 2 things: the total number of packets to be received and the number of the current packet in sequence. The former variable is called n and the later is called seq_number. We have randomly chosen a IPv4 identification number to be 112 for all packets. See figure 3.20.

```
#!/usr/bin/python

from scapy.all import *
import argparse
import random
import struct

def main():
    """
    """

def packet_with_seq_n(port, n, seq_number):
    packet = Ether(src="00:00:00:00:00:01", dst="00:00:00:00:00:02")/IP(src="10.10.10.1", dst="10.10.10.2", id = 112)/UDP(sport=7777,
dport=port)/Raw(load= struct.pack("ii", n, seq_number))
    sendp(packet, iface="eth0")

if __name__ == "__main__":
    main()
    parser = argparse.ArgumentParser(description='Simple script that sends TCP packets to an interface using scapy package')
    parser.add_argument('--dst-port', help='Destination port', type=int, action="store",    default=90)
    args = parser.parse_args()

    n =1000 #Total number of packets to be sent
    for seq_number in range(1, n+1):
        packet_with_seq_n(int(args.dst_port), n, seq_number)
```

Figure 3.20: Implementation of LossTest_UDPsend.py

In the controller.py we have written code to extract n and seq_number (mentioned previously) of each packets and make statistics out of it, e.g. the total number of packets not received by the controller and are lost. We also used CLI to access each registers of each switch in order to fetch the number of packets counts from the same identification number of 112. See figure 3.21.

```
        ####Code for TESTING LOSS MONITORING
        total = 0;
        current = 0;
        print "\nReady for loss test, please run LossTest_UDPsend.py from Host 1"
        while True:
            packetin = s3.PacketIn()        #Packet in!
            if packetin is not None:
                received += 1
                print "\n------PACKET IN received------"
            payload = packetin.packet.payload

            (total, current) = concatAndUnpack(payload, 'ii', 52, 59)
            print "Loss monitoring in progress: %f %%" % (current/float(total) *100)
            print "Total number of packets to be received: %d " % total
            print "Total number of packets actually received: %d " % received

            if((current/float(total) *100) == 100): #end of test
                break;
S
        receivedAtSwitch1 = subprocess.Popen(["python behavioral-model/targets/simple_switch/simple_switch_CLI.in \
                        --thrift-port 9090"], shell = True, stdout = subprocess.PIPE, stdin = subprocess.PIPE).communicate(input =
"register_read packet_counter 112")[0]

        receivedAtSwitch2 = subprocess.Popen(["python behavioral-model/targets/simple_switch/simple_switch_CLI.in \
                        --thrift-port 9091"], shell = True, stdout = subprocess.PIPE, stdin = subprocess.PIPE).communicate(input =
"register_read packet_counter 112")[0]

        receivedAtSwitch3 = subprocess.Popen(["python behavioral-model/targets/simple_switch/simple_switch_CLI.in \
                        --thrift-port 9092"], shell = True, stdout = subprocess.PIPE, stdin = subprocess.PIPE).communicate(input =
"register_read packet_counter 112")[0]

        receivedAtSwitch1 = int(filter(str.isdigit, receivedAtSwitch1[118:150]))
        receivedAtSwitch2 = int(filter(str.isdigit, receivedAtSwitch2[118:150]))
        receivedAtSwitch3 = int(filter(str.isdigit, receivedAtSwitch3[118:150]))

        print "\n\n****** SUMMARY LOSS MONITORING STATISTICS ******"
        print "Total number of packets to be received: %d " % total
        print "Total number of packets actually received by controller: %d " % received
        print "Total number of packets lost on the way (not received by controller): %d " % (total-received)

        print "Number of packets lost at SWITCH 1: %d " % (total-receivedAtSwitch1)
        print "Number of packets lost at SWITCH 2: %d " % (total-receivedAtSwitch2)
        print "Number of packets lost at SWITCH 3: %d " % (total-receivedAtSwitch3)
        print "Loss percentage at SWITCH 1: %f %%" % (((total-receivedAtSwitch1)/float(total))*100)
        print "Loss percentage at SWITCH 2: %f %%" % (((total-receivedAtSwitch2)/float(total))*100)
        print "Loss percentage at SWITCH 3: %f %%" % (((total-receivedAtSwitch3)/float(total))*100)
        print "****** END OF SUMMARY ******"
```

Figure 3.21: Implementation of loss monitoring testing in the controller.py

Now its time to test loss monitoring. First we run a control test with normal settings with sending 1000 UDP packets and capture the result. See figure 3.22

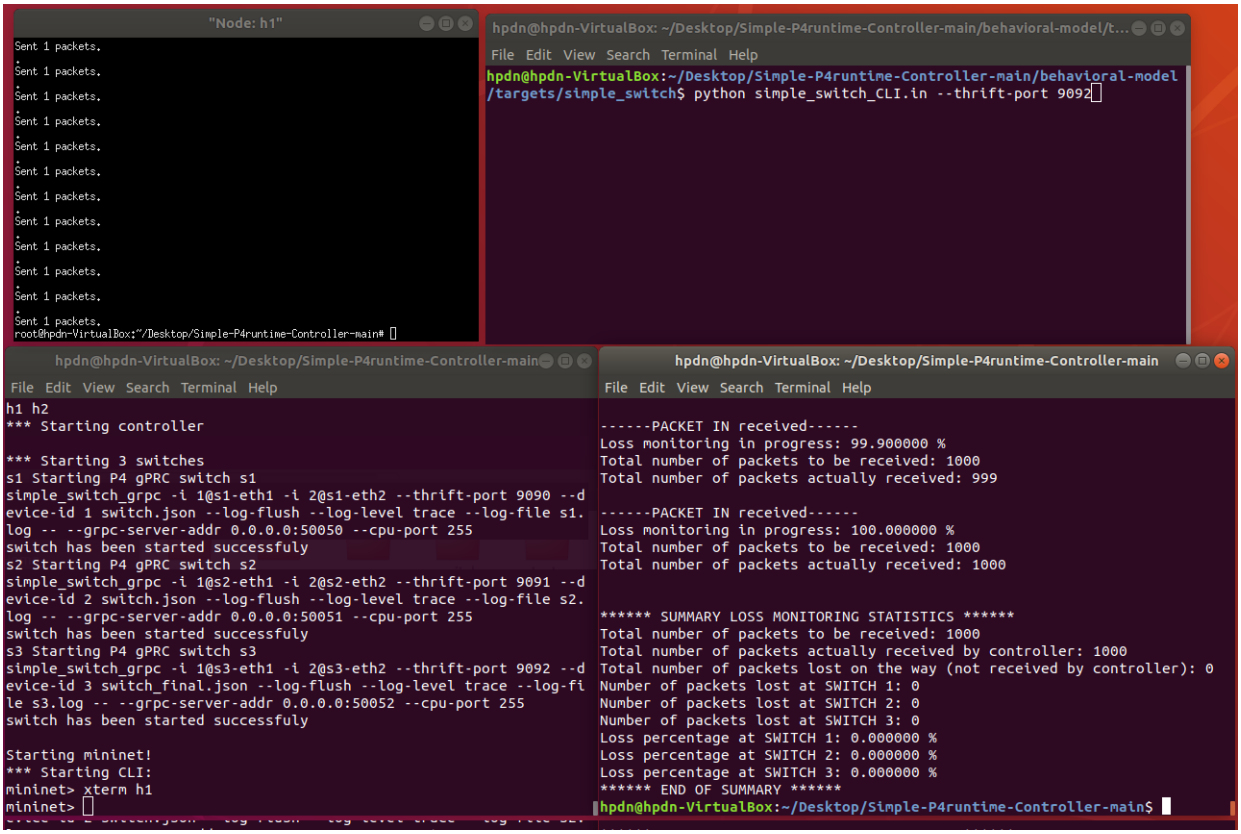


Figure 3.22: Control test

As you can see, as already expected and can be seen in a summary of loss monitoring statistics, there would be no packet loss at any switch under normal condition.

Now we introduce an artificial packet loss by simply increasing the queue processing rate of switch 2 with "set_queue_rate 10" and see what happens. Note that we only set the queue rate to 10 for only a couple of seconds before we set it back to maximum queue rate. See figure 3.23.

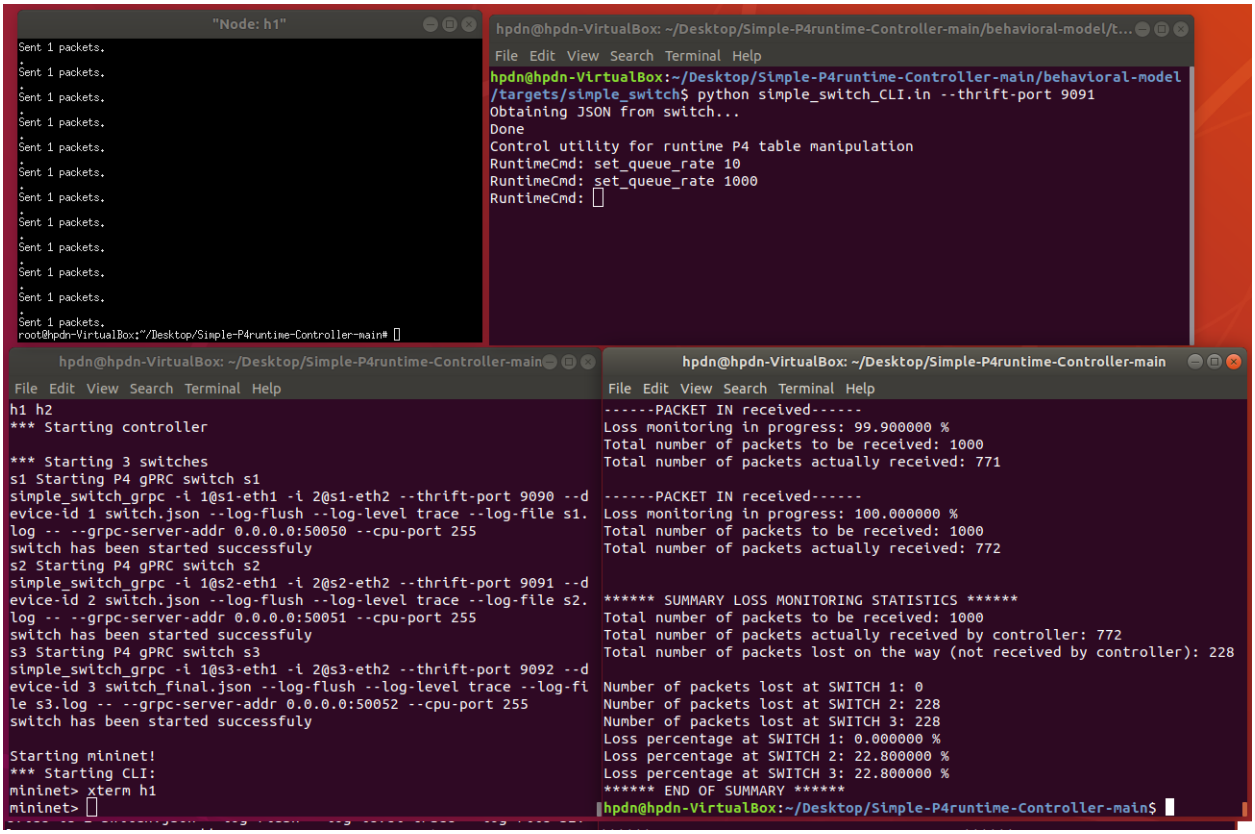


Figure 3.23: Loss test with induced loss at switch 2

In the summary you can see that there are 228 packets NOT dropped at switch 2 and therefore they did not arrive at switch 3 as well. This number matches the amount of packets that is NOT received by the controller. The loss monitoring test has correctly displayed the amount and percentage of loss packets in each switch. See figure 3.24 to see the result of a test where a lowered queue rate is temporary induced at switch 3.

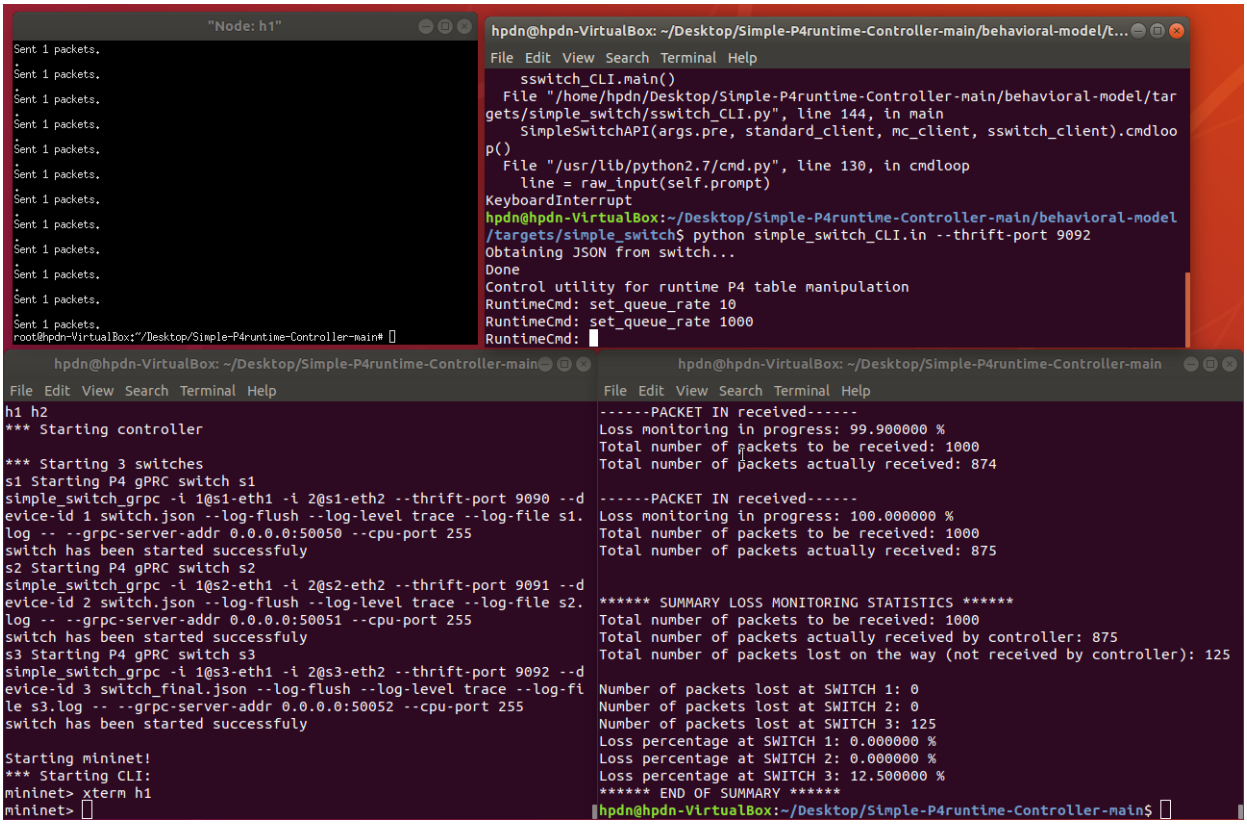


Figure 3.24: Loss test with induced loss at switch 3

As expected of correct loss monitoring, switch 3 is showed here to have dropped/not received 125 packets out of 1000 packets.

4

Results & Comparison

4.1. Openflow

4.1.1. Bandwidth monitoring

S. No.	**Configured Bandwidth (Mbps)**	**Maximum Throughput (Mbps)**	**Topology**	**Figure**
1.	20	19.606	Linear with two switches.	Fig 2.5
2.	20	18.5	Linear with five switches.	Fig 2.6
3.	20	19.2	Linear with five switches along with two clients and hosts	Fig 2.7
4.	40	38.1	Linear with two switches.	Fig 2.9
5.	40	37.5	Linear with five switches	Fig 2.10
6.	40	38.6	Linear with five switches along with two clients and hosts	Fig 2.11
7.	20	19.606	Ring with five hosts and five switches	Fig 2.8
8.	40	38.1	Ring with five hosts and five switches	Fig 2.12

Table 4.1: Bandwidth Results

- From Fig 2.6-2.7, it is evident that there are two flows, one from switch s2 to switch s1 and the second one is from s1 to s2. The size of the first flow is smaller compared to the second flow because the second flow is carrying the payload while the first flow consists of ACK messages as explained in CS4055 High Performance Data Networking Reader. The gradual increase in throughput demonstrates the TCP slow start algorithm which is a mechanism for congestion control. The initial sender's window size is smaller compared to receiver's window size. However, it doubles the rate every RTT until congestion is detected.
- From the Table 4.1, it can be clearly inferred that the maximum throughput which was measured by the monitoring app was close to the configured bandwidth but these two values were never equal. This observation stands true for all kinds of topologies and bandwidth configurations that were tested. The reason behind the marginal differences in the throughput value are latency,jitter and packet loss.
- The throughput also decreased as the number of switches were increased from two to five as observed in 2 and 5. After adding three more switches

between the source and destination, the latency increased which reduced the maximum throughput.

- Further, if we compare ring topology and linear topology (from 1,7 and 6,8) the maximum throughput remains the same because in our case ring topology acts as point-to-point linear topology.

4.1.2. Delay Monitoring

- The measured delay was found to be in milliseconds for every topology which was tested upon.
- The delay monitoring app was tested on different topologies such as linear and complete graph with varying nodes. The delay results were same for linear as well as complete mesh topology. The reasons for the unexpected delays were overhead due to Linux kernel and stacking of intermediate variables used for delay calculation as discussed in chapter 2.

4.1.3. Loss Monitoring

- The packet loss was greater than the assigned value of 5 percent with TCP traffic as observed in Fig 2.28 and equal to 0 percent for assigned value of 0 percent(Fig 2.26).
- For UDP traffic, the packet loss was higher than the assigned value and also higher when compared to the TCP traffic for the same assigned loss rate.
- The reason for higher packet loss is that UDP does not contain any congestion control mechanism (sliding window) which is present in TCP. As a result, the packets start dropping if there is congestion and UDP keeps sending the packets at the same rate.
- Another observation was that the bandwidth was also reduced due to the increase in packet loss percentage.

4.2. P4

4.2.1. Bandwidth monitoring

- The throughput achieved was equal to the assigned bandwidth when the rule to drop level 1 packet was disabled which was 1 Mbit per second. However, after configuring the rule to drop level 1 packet the achieved throughput got reduced to 800 Kbits per second.
- By using meters,the network bandwidth management becomes easier. If a flow becomes red, which means the rate of the packets arriving at the flow exceeds the specified rate and drop packet action is configured then the switch will start dropping packets which acts as a means of congestion control. In this manner, a better user experience is ensured.
- The meter also safeguards the network from packets of high data rates, that might hog the entire bandwidth of the network by limiting the rate of each flow.

- From Fig 3.15 and Fig 3.17, we can notice that since we have sent UDP traffic, the observed throughput was constant after uniform time intervals unlike TCP which used slow start algorithm for congestion control.
- Currently, we have limited the number of packets for each flow but this functionality can also be extended to limit the data rate which exceeds the specified limit.

4.2.2. Delay Monitoring

- The queuing delay was found to be in milliseconds, given in Fig 3.7. The queuing delay is the time a packet waits in the queue before getting processed.
- The delay monitoring app measured the correct queuing delay of all three switches. This delay information can be used to detect which node in the network is taking longer time to process the packets. Then, bottlenecks in the network can be avoided by finding out the root cause, for example replacing an outdated switch that has very slow processing time. These delays also affect the network throughput.
- Delay Monitoring can be used in the future to find out if a particular path is slow and switch to other paths that have faster processing switches.

4.2.3. Loss Monitoring

- From Fig 3.22, we can observe that the number of lost packets were 0, which means that there were no network issues such as bandwidth unavailability, jitters and increased latency.
- From Fig 3.23, decrease in queue processing rate resulted in small packet loss, as it introduced a larger queuing delay in the network.
- If there is a packet loss at one switch(s1) then the next few switches will also suffer packet loss because there will be no forwarding from the switch S1.

4.3. OpenFlow and P4 Comparison

4.3.1. Openflow monitoring approach

- For measuring network statistics, OpenFlow follows active approach which means injecting additional packets for collecting network statistics.
- The drawback of this approach is that it does not give real-time network statistics.In addition,these probe packets generate additional traffic which takes up extra bandwidth and storage.
- The polling interval should be small for getting updated statistics from time to time. Even if the polling frequency is in seconds, microsecond congestion cannot be avoided.

4.3.2. P4 monitoring approach

- The P4 monitoring follows a different approach. The monitoring app works by creating additional headers which are encoded in the data packet itself.
- The drawback of this approach is the increased packet size due to extra headers.
- The increase in packet size can be a problem for networks which consist of large number of nodes as these will consume more bandwidth.
- Also cloning was done to replicate packets which were sent towards the controller which acted as the monitoring node and aggregated network statistics. Together these cloned packets and additional headers can take up significant bandwidth.

From above, it follows that both these discussed approaches have various advantages and disadvantages. The P4 approach is more efficient for smaller networks consisting of fewer number of nodes.

4.3.3. Comparison based on Overhead

- For OpenFlow bandwidth monitoring and loss monitoring, first and last switch were regularly polled by sending stats request packets. The polling frequency was chosen as 2 seconds. Similarly, additional probe packet was created to measure the end-to-end delay. These packets are responsible for the extra overhead.
- For delay monitoring, a custom header was created for aggregating the latency statistics per hop. For bandwidth monitoring, configuring the meter with the required rate was enough for bandwidth management along with addition of header for meter in the beginning of the code. For loss monitoring, register was used to keep track of the data fragments and later these statistics were collected from the monitoring node.
- Consider the latency monitoring in which the data packet became 32 bit long after the last hop. This result is shown in fig This increase in packet size will not be suitable for networks which consists of larger number of nodes since with every hop the packet size will considerably increase.

```
}

header latency_t{
    bit<32>  queuing_delay;
}
```

Figure 4.1: latency header

From above observations, it can be inferred that OpenFlow overhead is greater as compared to the overheads in P4. Instead of creating additional probe packets, telemetry information is embedded inside the header which is less compared to the OpenFlow overhead.

4.3.4. Comparison based on Network Statistics

The P4 and OpenFlow are also compared based on the following parameters:

- **Bandwidth monitoring**: In P4 the throughput was equal to the configured bandwidth whereas in OpenFlow the measured throughput was always less than the set limit.
- **Loss monitoring**: The packet loss measured was similar for both P4 as well as OpenFlow.
- **Delay monitoring** :In P4 delay monitoring, we have measured only queueing delay and not the end-to-end delay between the switches because queueing delay information is useful in locating the switch which has gone out of order. This can be used in future to find the congestion bottleneck. In Openflow,end-to-end delay was measured which takes into account propagation delay, trasmission delay and queueing delay.

5

Conclusion

The goal of this project was to do a comparative study of OpenFlow monitoring application with P4 monitoring application. This comparison was based on bandwidth, delay and packet loss. For carrying out the comparison, we first developed applications both for P4 as well as OpenFlow for monitoring network statistics. Then we analysed the results from each one of these apps and compared those results with iperf results for validation. The approach followed for collecting data in OpenFlow and P4 were different from each other. After implementing these monitoring functionalities, both the apps were thoroughly tested with varying traffic, topologies and link bandwidths.

Based on these test results, it can be concluded that P4 is more efficient compared to the OpenFlow as the overhead is less, it provides flexible data plane, reduces the load on controller to a great extent and achieved maximum throughput. However, for monitoring in large-sized networks, especially with latency monitoring, the overhead might increase due to additional headers.

Bibliography

[1] Randeng works. URL https://carolinafernandez.github.io/development/2019/08/06/Recurrent-processing-in-P4.

[2] Gary Lee. *Cloud networking: Understanding cloud-based data center networks*. Morgan Kaufmann, 2014.

[3] D. Marschke, J. Doyle, and P. Moyer. *Software Defined Networking (SDN): Anatomy of OpenFlow Volume I*. Number v. 1. Lulu Publishing Services, 2015. ISBN 9781483427249. URL https://books.google.co.in/books?id=ecM5CgAAQBAJ.

[4] p4lang. p4lang/behavioral-model, . URL https://github.com/p4lang/behavioral-model/tree/main/targets/simple_switch_grpc#enabling-the-thrift-server.

[5] p4lang. p4lang/behavioral-model, . URL https://github.com/p4lang/behavioral-model/blob/main/docs/simple_switch.md.

[6] p4lang. p4lang/tutorials, . URL https://github.com/p4lang/tutorials/blob/master/exercises/source_routing/solution/source_routing.p4.

[7] Niels LM Van Adrichem, Christian Doerr, and Fernando A Kuipers. Opennetmon: Network monitoring in openflow software-defined networks. In *2014 IEEE Network Operations and Management Symposium (NOMS)*, pages 1–8. IEEE, 2014.